In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

University of Alberta

Utilizing domain dependent knowledge for planning in answer set programming

by

Xiumei Jia

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2003

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-87790-6
Our file Notre référence
ISBN: 0-612-87790-6

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canada

University of Alberta

Library Release Form

Name of Author: Xiumei Jia

Title of Thesis: Utilizing domain dependent knowledge for planning in answer set programming

Degree: Master of Science

Year this Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Xiumei Jia
Edmonton, Alberta
Canada

Date: May 30, 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Utilizing domain dependent knowledge for planning in answer set programming** submitted by Xiumei Jia in partial fulfillment of the requirements for the degree of **Master of Science**.

Jia You
Supervisor

Guohui Lin

Marek Reformat
External Examiner

Date: May 29, 2003

Abstract

Research on planning has two components: representation and efficient plan generation. Representation is to specify domain independent knowledge by using a language with well-defined semantics. Efficient plan generation lies in how to prune search space during the search for plans. Logic programming with the stable model semantics, which provides a framework for *answer set programming*, has been suggested as a new paradigm for solving planning problems. The idea is that the action predicates that are true in a stable model of the encoding of a planning problem constitute a valid plan of the underlying planning problem.

Planning is computationally difficult. The complexity of the planning problem is known to be PSPACE-complete for finite domains [6, 9]. By fixing the length of plans, the planning problem is reduced to be NP-complete [35].

Although it is commonly agreed that domain dependent knowledge can improve search performance significantly, it has not been investigated extensively that domain dependent knowledge could improve search efficiency in the specific context where the planning problems are encoded in logic programming based on the stable model semantics.

Our goal in this thesis is to investigate the utilization of domain dependent knowledge for planning problems which are encoded in logic programming. To make use of the domain dependent knowledge, we need to extract knowledge from the planning domain. In this thesis, a classification is first proposed which can provide a guide to extract domain dependent knowledge. We are aware of no previous work in the literature that classifies domain dependent knowledge. For each class of domain dependent knowledge, a standard encoding is provided, so that the knowledge can be added to the encoding of the domain independent knowledge as extra constraints that can be used to prune search space for planning.

A number of experiments have been conducted on problems taken from the planning literature. The experimental results show that domain dependent knowledge encoded

as extra constraints can improve search efficiency under the specific context where a planning problem is represented in logic programming.

To My Family
and My Friends

Acknowledgements

Thanks to my supervisor Prof. You for providing guidance and advice not only on technical aspects but also on effective writing.

I am grateful to the members of my examining committee for their valuable comments, suggestions and time spent reading the thesis.

I would like to thank my dear husband Chun for his love, support and encouragement. Special thanks to all my friends for the various forms of help and encouragement that they have given me.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Thesis layout . . . 4

2 Background **5**
- 2.1 The stable model semantics . . . 5
- 2.2 Domain-restricted function-free logic program . . . 7
- 2.3 Example programs . . . 8
- 2.4 Introduction to Smodels . . . 10
  - 2.4.1 The decision procedure . . . 10
  - 2.4.2 Function expand(P,A) . . . 11
  - 2.4.3 Function lookahead(P,A) . . . 15
  - 2.4.4 Function heuristic(P,A) . . . 15
- 2.5 Extended rules in logic programming . . . 16

3 Encoding of Planning in Logic Programming **18**
- 3.1 Action theory and planning . . . 18
- 3.2 Representing fluents and actions in logic programming . . . 18
- 3.3 Representing initial and goal states in logic programming . . . 19
- 3.4 Representing action systems in logic programming . . . 19
- 3.5 Encoding of the gripper problem . . . 22
  - 3.5.1 Encodings of initial and goal states . . . 24
  - 3.5.2 Encoding of the action system . . . 24

4 Previous Work on Domain Dependent Knowledge **29**
- 4.1 CPlan . . . 29
- 4.2 TLPlan . . . 31

5 Domain Dependent Knowledge in Logic Programming **33**
- 5.1 Classification of domain dependent knowledge . . . 33
- 5.2 End state knowledge . . . 35
- 5.3 State relationship knowledge . . . 36
  - 5.3.1 Relationships about state properties . . . 36
  - 5.3.2 Relationships among state conditions and actions . . . 37
- 5.4 Procedural knowledge . . . 38
  - 5.4.1 Total procedural knowledge with known state distance . . . 39
  - 5.4.2 Partial procedural knowledge with known state distance . . . 41
  - 5.4.3 Procedural knowledge with unknown state distance . . . 42
  - 5.4.4 Procedural knowledge with unknown objects . . . 43
- 5.5 Symmetry knowledge . . . 44
  - 5.5.1 Symmetry discovery . . . 45
  - 5.5.2 Symmetry breaking . . . 46
- 5.6 Distance knowledge . . . 49
- 5.7 Related work on domain dependent knowledge . . . 50

6 Experiments **52**
6.1 Five planning problems 52
6.2 Experiments in applying end state knowledge 55
6.3 Experiments in applying state relationship knowledge 59
6.4 Experiments in procedural knowledge 64
6.5 Experiments in symmetry knowledge 70
6.6 Experiments in distance knowledge 75
6.7 Summary of the experimental results 83

7 Conclusion and Future Work **85**
7.1 Conclusions 85
7.2 Future work 85

A Smodels' Syntax **86**
A.1 Constant,Variable and Literal 86
A.2 Rules 86
A.3 Statements 86

B Encodings of the Blocks World Problem in Smodels **88**

C Encodings of the Logistics World in Smodels **90**

D Encoding of the Ferry Problem in Smodels **94**

E Encoding of the Elevator Problem in Smodels **96**

List of Figures

1	Algorithm of *smodels*	10
2	Function expand(P,A)	11
3	Function lookahead(P,A)	15
4	An example for the gripper problem	23
5	An example for distance constraint in the logistics problem	30
6	An example to show symmetry breaking can prune search space	44
7	An example of goal position in the blocks world problem	60
8	Procedural knowledge prunes search space	66
9	Construction of distance knowledge for the gripper problem	76

List of Tables

1 An optimal plan for the gripper problem instance in Figure 4 23
2 Experimental results of the gripper problem with/without ESK . . . 56
3 Experimental results of the logistics problem with/without ESK . . . 56
4 Experimental results of the elevator problem with/without ESK (have solution) . . . 57
5 Experimental results of the elevator problem with/without ESK (no solution) . . . 57
6 Experimental results of the ferry problem with/without ESK (have solution) . . . 58
7 Experimental results of the ferry problem with/without ESK (no solution) . . . 58
8 Experimental results of applying $ESK09$ to the blocks world problem 59
9 Experimental results of applying SRK01 to the blocks world problem 60
10 Experimental results of applying $SRK02$ to the blocks world problem 61
11 Experimental results of applying $SRK03$ to the logistics problem . . 62
12 Experimental results of the gripper problem with/without SRK . . . 63
13 Experimental results of the ferry problem with/without SRK (have solution) . . . 63
14 Experimental results of the elevator problem with/without SRK . . . 64
15 Experimental results of the gripper problem with/without PK (have solution) . . . 66
16 Experimental results of the gripper problem with/without PK (no solution) . . . 67
17 Experiment results of the ferry problem with/without PK (have solution) . . . 68
18 Experimental results of the ferry problem with/without PK (no solution) 68
19 Experimental results of the logistics problem with/without PK (have solution) . . . 69
20 Experimental results of the elevator problem with/without PK (have solution) . . . 69
21 Experimental results of the elevator problem with/without PK (no solution) . . . 70
22 Experimental results of the gripper problem with/without SK (have solution) . . . 72
23 Experimental results of the gripper problem with/without SK (no solution) . . . 72
24 Instances of the ferry domains . . . 73
25 Experimental results of the ferry problem with/without SK (have solution) . . . 73
26 Experimental results of the ferry problem with/without SK (no solution) 74
27 Experimental results of applying symmetry knowledge to the logistics problem (have solution) . . . 74
28 Experimental results of applying symmetry knowledge to the logistics problem (no solution) . . . 74
29 Experimental results of the gripper problem with/without DK (have solution) . . . 76

30 Experimental results of the gripper problem with/without DK (no solution) 77
31 Experimental results of applying DK to the logistics problem 78
32 Experimental results of applying DK to the logistics problem 79
33 Experimental results of applying DK to the blocks world problem (have solution) 80
34 Experimental results of applying DK to the blocks world problem (no solution) 81
35 Experimental results of the ferry problem with/without DK (have solution) 81
36 Experimental results of the ferry problem with/without DK (no solution) 82
37 Experimental results of the elevator problem with/without DK (have solution) 82
38 Experimental results of the elevator problem with/without DK (no solution) 82

1 Introduction

A planning problem can be described as follows: Given

1. a set of (possibly) applicable actions with or without costs,
2. an initial state, and
3. a goal state,

find a plan (sequence of actions) which leads from the initial state to the goal state.

Each action may be associated with a cost. The cost of a plan is the total cost of all actions in the plan. An optimal plan is a plan with minimal cost. If every action has a unit cost, the optimal plan is the shortest plan. In this thesis, we only consider the shortest plan as the optimal plan.

1.1 Motivation

Research on planning has two components: representation and efficient plan generation. Representation is about specifying domain independent knowledge using a language with well-defined semantics. Efficient plan generation lies in how a plan can be generated efficiently and how to prune search space.

The planning problem has traditionally been formalized as one of logic deduction. The best-known logical formalization of planning is STRIPS [11]. In this system, a state is represented by a knowledge base of fluents, and a set of operators is used to change the state. A fluent is a property which could be *true* or $false$ for some state. For example, $at(robot, room_i)$ represents that the robot is at room $room_i$. For each action, STRIPS uses four operators to describe under what preconditions the action should be executed, and what the effects are after undertaking the action. These four operators are: *Action*, *Add*, *Delete*, and *Precondition*. *Action* specifies the name of an action, *Add* tells what becomes true after an action, *Delete* tells what ceases to be true after an action, *Precondition* specifies what must be true to undertake an action.

For example, consider the gripper problem where a robot is used to move all the balls in one room to another. The following code shows that when the robot is at room m, it can take action $goto(m, n)$. After taking the action, *Add* operator will add $at(robot, n)$ to the add list, which means that $at(robot, n)$ becomes *true*. The *Delete* operator will include $at(robot, m)$ in the delete list ($at(robot, m)$ becomes $false$).

```
Initial state: at(robot,m)
Goal: at(robot,n)

Operators:

  Action:       goto(m,n)
  Add:          at(robot,n)
  Delete:       at(robot m)
  Precondition:at(robot,m)
```

In recent years, logic programming has become a major vehicle for the study of knowledge representation and reasoning. The simple syntactic requirement makes it easier not only to reveal the principles but also to generate realistic, implementable systems. It is extended from Horn clause logic programming by allowing negation as failure. Logic programming based on the stable model semantics [14] is particularly suitable for solving constraint problems. In particular, it has been used for solving planning problems [15]. The idea is that the action predicates that are true in a stable model of the encoding correspond to a valid plan of the underlying planning domain. Recent implementations of the stable model semantics include SLG [7], Smodels [36] and DLV [10].

Answer set programming is a new programming paradigm. In this paradigm, a problem is represented by a program specifying the constraints that must be satisfied. A program may have zero, one or more intended models. These models are called answer sets, each of which corresponds to a solution of the problem being solved. When there is no answer set for the given program, that means the underlying problem has no solution. Logic programming based on the stable model semantics provides an answer set programming system. In this system, a problem is represented by a logic program, and a stable model of the logic program corresponds to a solution of the problem.

Planning is computationally difficult. It is PSPACE-complete to determine if a given planning instance has any solutions [6, 9]. By fixing the length of plans, the planning problem can be reduced to be NP-complete [35].

Search control is identified to be useful when solving the planning problem. There are two types of search control: knowledge-based search control and search heuristics. Knowledge-based search control relies on domain dependent knowledge. This is the key difference between the knowledge-based search control and various search heuristics. Although search heuristics could provide impressive improvements in performance, there is a growing belief that domain dependent knowledge may be the key to future performance gains [40].

Many planning domains could offer additional specific structure and/or other domain dependent knowledge that can guide the search, and thus ease the difficulty of planning. The SOAR system [21] was the first to utilize domain dependent knowledge with forward chaining. A refined version of this system becomes a prominent part of the PRODIGY system [39]. TLPlan [1] and TALplan [8] are based on STRIPS to encode the domain independent knowledge. In order to add domain dependent knowledge into search, a new language is constructed specifically to encode this knowledge. They experiment a significant speedup after applying domain dependent knowledge.

CPlan [4] is a planning system which encodes a planning problem as a constraint satisfaction problem (CSP). It exploits certain types of domain dependent knowledge and provides impressive performance. When specifying domain dependent knowledge as constraints, the user needs to provide information on what kind of search strategy should be applied to these constraints. These search strategies include for-

ward checking, node consistency checking, and other techniques. Therefore, in this case, the user must have intimate knowledge of the underlying search algorithm.

Although it is commonly agreed that domain dependent knowledge can improve search performance significantly, it has not been investigated extensively whether domain dependent knowledge can improve search in the specific context where the planning problems are encoded as logic programs based on the stable model semantics. To utilize the domain dependent knowledge, we first need to extract the knowledge from the planning domains. Classification of the knowledge can provide a guide to extracting domain dependent knowledge. We are aware of no previous work in the literature that has classified domain dependent knowledge for planning.

In this thesis, first, a methodology is presented to encode planning problems in logic programming. Then, the domain knowledge of the planning problems is classified into five categories. In addition, in order to make use of the domain dependent knowledge, a formalization and a standard encoding are provided for each class. The idea is that the domain dependent knowledge is encoded as extra constraints. These constraints are added into the original encoding of the planning domain in order to prune search space.

The approach to making use of domain dependent knowledge proposed in this thesis differs from previous works. First, domain dependent knowledge will not be encoded into the underlying search algorithm; that is, the information we use does not make reference to the underlying planning algorithm; rather, it only changes the properties of the planning domain. This means that although the control information we utilize is domain dependent, the user need not have intimate knowledge about the underlying planning algorithm.

Second, in contrast with TLPlan and TALplan, we need not build a new language for the encoding of domain dependent knowledge. The domain dependent knowledge is encoded as extra constraints into the original program which encodes the domain independent knowledge of the planning problem. These extra constraints are constructed in the same syntax as the original program.

A number of experiments are conducted on problems taken from the planning literature. For these experiments we used Smodels [36], a recent efficient implementation of the stable model semantics that seems to outperform other existing systems. It turned out that encoding domain dependent knowledge as extra constraints is an effective way to improve search efficiency. Every class of domain dependent knowledge could provide some way to prune search space.

The contributions of this thesis are as follows:

1. A classification of domain dependent knowledge is proposed. The classification can be used as a guide for users in extracting domain dependent knowledge from planning domains.

2. A standard encoding for each class of the domain dependent knowledge is presented. The encodings provide a way to translate the knowledge as extra constraints and incorporate these constraints into the logic programs which represent underlying planning problems.

3. Experiments on a set of planning problems are conducted. The experimental results show that domain dependent knowledge encoded as extra constraints can improve search performance significantly for the planning problems encoded in logic programming.

1.2 Thesis layout

Chapter 2 reviews some of the concepts in logic programming based on the stable model semantics. Also in this chapter, the Smodels system, an efficient implementation of the stable model semantics, is introduced. Chapter 3 provides a methodology to encode planning in logic programming. The gripper problem, a benchmark in the planning literature, is encoded to demonstrate the methodology. Chapter 4 discusses some previous work on domain dependent knowledge. Chapter 5 describes a classification of domain dependent knowledge. For each class of the knowledge, a formalization and standard encoding is provided in order to encode domain dependent knowledge into a logic program which represents a planning problem. Chapter 6 provides experimental results comparing the search performance of planning with and without domain dependent knowledge. The results show that domain dependent knowledge constructed using our encodings can improve search efficiency significantly. Chapter 7 presents conclusions and points out some future work.

2 Background

In this chapter, the basic concepts and implementations of logic programming based on the stable model semantics are reviewed. First, in Section 2.1, the stable model semantics for logic programs are explained. Then, in Section 2.2, domain-restricted function-free logic programs with variables are introduced. In Section 2.3, we take two problems, the colorability problem and the Hamiltonian Cycle problem, as examples to demonstrate logic programming based on the stable model semantics. In Section 2.4, the Smodels system, which is an efficient implementation of the stable model semantics, is presented in detail. The reason to provide the implementation detail of the Smodels system is that our experiments are conducted on the system, and it is beneficial for readers to understand the implementation. However, the readers can still understand the rest of the thesis if they choose to omit this section. Finally, in Section 2.5, a few extended rules of logic programming are introduced.

2.1 The stable model semantics

Stable models were introduced to provide semantics for logic programs with negation.

We first assume a propositional language in order to explain the semantics and concepts clearly. Later on, we will define the semantics for logic programs with variables.

A *normal logic program* is a set of rules of the form

$$h \leftarrow a_1, ..., a_n, not\, b_1, ..., not\, b_m.$$

where h, $a_1, ..., a_n$, $b_1, ..., b_m$ are propositional atoms. An expression such as *not* b is called a *not-atom*. Atoms and not-atoms are referred to as *literals*. Not-atoms are also called *negative* literals and atoms are called *positive* literals. In the above rule, atom h is called the head of the rule and the other literals make up the body.

A logic program without not-atoms is called a *positive* program (or a *negation free* program). A model of a logic program is a truth value assignment which satisfies all the rules in the program. It is well-known that a positive program always has a unique minimal model which can be computed iteratively. A model of a positive program is minimal if there is no proper subset of it which is a model of the program.

The unique minimal model coincides with the least fixpoint of the function T_P defined as follows:

$$T_P(S) = \{h \mid h \leftarrow a_1, ..., a_n \in P,\ \{a_1, ..., a_n\} \subseteq S\}$$

where S is a set of atoms and P is a positive program. The function is monotonic since given sets of atoms S_1 and S_2, if $S_1 \subseteq S_2$ then $T_p(S_1) \subseteq T_p(S_2)$. The least fixpoint of T_P can be constructed iteratively as follows:

$$T_p^{\uparrow 0} = \emptyset$$
$$T_p^{\uparrow (i+1)} = T_p(\, T_p^{\uparrow i}\,) \ when\ i \geq 0$$

When for a i where $i \geq 0$ and $T_p(\ T_p^{\uparrow i}\ ) = T_p^{\uparrow(i)} = S$, then S is a fixpoint. S is a least fixpoint if for every fixpoint W, $S \subseteq W$.
Let P be a normal logic program. The notion of a stable model for P is defined as follows [14]:
For any set of atoms A from P, let P^A be the program obtained from P by:

1. deleting each rule in P that has a not-atom *not* x in its body such that $x \in A$, and by
2. deleting all not-atoms in the remaining rules.

P^A is called the *reduct* of P with respect to atom set A. Clearly, P^A is a positive program, therefore P^A must have a unique minimal model M.

Definition 2.1 An atom set A is a *stable model* (also called an *answer set*) of normal logic program P if and only if it coincides with the minimal model of P^A.

Example 2.1 Let normal logic program P be:

$a \leftarrow not\ b.$
$b \leftarrow not\ a.$

$M_1 = \{a\}$ and $M_2 = \{b\}$ are the only two stable models for program P. The reduct of P^{M_1}, for example, is obtained as follows:

1. deleting *not* b from the first rule to get $\{a \leftarrow . \ b \leftarrow not\ a.\}$, and
2. deleting the second rule

We then get $P^{M_1} = \{a \leftarrow\}$. As there is only one rule with an empty body in the program, the minimal model is $\{a\}$, which is precisely M_1. So $M_1 = \{a\}$ is a stable model of program P. Similarly, one can verify that $M_2 = \{b\}$ is also a stable model.

$M_3 = \{a, b\}$ is not a stable model, since P^{M_3} is empty, therefore, the minimal model of P^{M_3} is also empty, which does not coincide with M_3.

All stable models are *justified* in the sense that every atom in a stable model has to have some reason to be there. That is, if an atom is in a stable model, there must be a rule with the atom as the head where all the body literals are satisfied. This means that all the positive literals in the body are in the stable model while all the negative literals are not in the stable model. The third model $M_3 = \{a, b\}$ in Example 2.1 is not justified because both rules have unsatisfied bodies.

A program may have no stable model. For example, let program P be

$a \leftarrow not\ a.$

There is no stable model for P. There are only two possible models: $\{a\}$ and $\emptyset$. If $\{a\}$ is a model, then the body of the only rule in program P, *not* a, is $false$, which does not support a. While $\emptyset$ is not a model, since it does not satisfy the rule. Therefore, any logic programs containing rules like this have no stable model.

A constraint in logic programs is of the following form

$$\leftarrow a_1, ..., a_n, not\, b_1, ..., not\, b_m.$$

A model M satisfies the constraint if the body of the constraint is $false$ in M. If M is a stable model of a program, then M must satisfy all the constraints in the program.

Under the stable model semantics, the above constraint can be translated to the following rule:

$$f \leftarrow not\, f,\ a_1,\ ...,\ a_n,\ not\, b_1, ..., not\, b_m.$$

where f is a new atom. This translation is equivalence-preserving. Let program P_c be the program that contains the constraint, and P_r be the program of P_c where the constraint is replaced by the above rule. It can be shown that if a model M is a stable model of P_c, then M is also a stable model of P_r, and vice versa. The reason is that the body of the constraint must be $false$ in M for both cases. Because of this, we will feel free to use them under the stable model semantics.

The above definition of the stable model semantics is not constructive and can only be used to verify whether or not a set of atoms is a stable model of a program. In fact, the problem of determining whether a normal logic program has a stable model is NP-complete [23].

2.2 Domain-restricted function-free logic program

In general, we write programs with variables. Given the condition that a program is function-free, the stable models of the program with variables are the stable models of its corresponding instantiated, ground program.

A Function-free normal logic program with variables [30] refers to sets of rules of the form

$$\begin{aligned} h(V_1, ...) \leftarrow\ & a_1(X_{11}, ..), ..., a_n(X_{n1}, ...,), \\ & not\ b_1(Y_{11}, ...), ..., not\ b_m(Y_{m1}, ...). \end{aligned}$$

where h, a_i, b_j are predicates, and V_i, $X_{i,j}$, $Y_{i,j}$ are variables or constants.

The absence of functions in a function-free logic program controls the explosion of the size of the instantiated program, and guarantees that the instantiated ground program is finite.

From now on, when we refer to a logic program, we mean a function-free normal logic program.

Domain-restricted logic programs form a subclass of logic programs with variables. The subclass is defined by the property of domain-restrictedness. The property of domain-restrictedness is defined as follows:

Definition 2.2 [30] Let P be a logic program with variables and D be a set of predicates. Then P is *domain restricted* with respect to D, if for each rule, it holds that every variable appearing in the rule appears also in a positive body literal for which the predicate is from D.

We call the predicates in D *domain predicates*. A domain predicate restricts the range over which a variable can obtain values. The domain predicate also restricts the instantiation of the rule with the predicate. The values a variable could obtain are the values that make domain predicates *true*.

Suppose X is the only variable in the following rule, and *time* is the only domain predicate:

$$h(X) \leftarrow time(X),\ not\ body(X).$$

The values that X could be assigned in this rule are the only values that make $time(X)$ *true*. For example, if

$time(1)$.
$time(2)$.

are in the same program, the above rule is equivalent to the following two ground rules:

$h(1) \leftarrow time(1),\ not\ body(1)$.
$h(2) \leftarrow time(2),\ not\ body(2)$.

2.3 Example programs

In this section, we will use two well-known problems, the colorability problem and the Hamiltonian cycle problem, to illustrate logic programming based on the stable model semantics.

Colorability

The problem is, given facts about vertices, arcs and available colors, find an assignment of colors to vertices such that each vertex has a color and two vertices connected with an arc do not share the same color. The following is a logic program that solves the colorability problem [19].

```
% rules to generate answer sets

color(V,C) ← vertex(V), col(C),not otherColor(V,C).
otherColor(V,C)← vertex(V), col(C), not color(V,C).

%utility rule

hasColor(V)← vertex(V), col(C), color(V,C).

%Constraints

← arc(V,V1),color(V,C),color(V1,C), V ≠ V1,
  vertex(V), vertex(V1),col(C).

← not hasColor(V), vertex(V).

← color(V,C),color(V,C1),col(C),col(C1),vertex(V).
```

In the above program, *col* and *vertex* are domain predicates. $color(V, C)$ means that vertex V gets color C. The first two rules are used to generate all possible assignments of colors to vertices. If $color(V, C)$ is in a stable model, the second rule specifies that $otherColor(V, C)$ can not be in the same stable model. On the other hand, if $otherColor(V, C)$ is in a stable model, the first rule determines that $color(V, C)$ can not be in the same model. The auxiliary predicate $otherColor(V, C)$ is used to provide all assignments where $color(V, C)$ gets value $false$.

The third rule and the second constraint specify that every vertex must get at least one color. If there is a vertex V, $color(V, C)$ are $false$ for all colors in a truth value assignment, then this assignment is not stable model. The first constraint specifies that two vertices connected with an arc should not get the same color. If an assignment contains atoms like $color(u, green)$ and $color(v, green)$ where u and v are two distinct vertices, this assignment can not satisfy the second constraint, and thus is not a stable model. The third constraint is used to eliminate models where one vertex gets more than one color.

Hamiltonian Cycle

The problem is, given a set of facts defining the vertices and edges of a directed graph and a starting vertex v_0, find a path which visits every vertex exactly once, and then closes at the starting vertex. These kind of paths are called Hamiltonian cycles. The answer sets for the following program correspond to the Hamiltonian cycles of a given graph.

```
in(U,V)← edge(U,V), not out(U,V).
out(U,V)← edge(U,V),not in(U,V).

reachable(V) ← in(v0,V).
reachable(V) ← reachable(U),in(U,V).

← in(U,V), in(U,W), V ≠ W.
← in(U,W),in(V,W), U ≠ V.

← vertex(U), not reachable(U).
```

That $in(U, V)$ is in a model means that $edge(U, V)$ is used. The predicate $out(U, V)$ means that $edge(U, V)$ is not used. The first two rules provide all the subsets of the edges in the given graph.

That $reachable(V)$ is in a model means that there is a path from v_0 to vertex V. This predicate is used to construct paths from v_0 to other vertices, until closing at v_0 again using edges chosen by the first two rules. The third and forth rules describe the following logic: If vertex V is connected directly to v_0 by an edge, it is reachable. Otherwise, if it is connected directly to another vertex, say U, which can be reached by a path from v_0, vertex V is also reachable.

The fifth and sixth rules are constraints, which prohibit that a vertex is visited more than once. For three distinct vertices, U, V and W, that predicates $in(U, V)$ and $in(U, W)$ are *true* in a model, means that $edge(U, V)$ and $edge(U, W)$ are used. It also means that vertex U is visited more than once. Similarly, if a vertex has two incoming edges in a path, that also means the vertex is visited more than once. The last constraint is used to specify that every vertex should be reached from v_0.

```
Function smodels(P,A)

A := expand(P,A)
A := lookahead(P,A)

If conflict(P,A) then
   Return false
Else if A covers Atoms(P) then
  Return true {A+ is stable model}
Else
  x := heuristic(P,A)
  If smodels(P,A⋃{x}) then
    Return true
  Else
    Return smodels(P,A⋃{not x})
  End if
End if
```

Figure 1: Algorithm of *smodels*

2.4 Introduction to Smodels

The Smodels system is an implementation of the stable model semantics for logic programs. This system consists of two components: *lparse* [37] and *smodels* [26, 27, 31]. *lparse* is a front-end that takes a domain restricted logic program as input and generates a ground logic program. Usually it produces a subset of the fully instantiated ground instance of the program. The subset is sufficient in the sense that the stable models are preserved. *smodels* works with this ground program and computes the stable models for it using a Davis-Putman like procedure [33]. The algorithm of *smodels* is based on backtrack search.

We first give the notations used in the rest of the section. Let A be a set of atoms and B be a set of literals. B *covers* A if $A \subseteq \{x |\ x \in B$, *or* $(not\ x) \in B\}$. B^+, B^- are sets of atoms where $B^+ = \{a |\ a \in B$ *and* a *is an atom*$\}$ and $B^- = \{a |\ not\ a \in B$ *and* a *is an atom*$\}$. For a program P, Atoms(P) denotes the set of all atoms x such that either x or *not* x appears in P, NAnt(P) denotes the set of all atoms x such that *not* x appears in P. For a set of literals B, $not(B) = \{not\ (x) \mid x \in B\}$. Atom set A *agrees with* B if $B^+ \subseteq A$ and $B^- \bigcap not(A) = \emptyset$. The *negation* of literal x is defined as $not(x)$ where if x is an atom, $not(x) = not\ x$, $not(not\ x) = x$.

2.4.1 The decision procedure

In the heart of *smodels'* algorithm is the construction of a set of atoms A which represents a partial truth assignment. An atom $a \in A$ means that a is assigned value *true* while a negative literal *not* $b \in A$ means b gets value $false$. If A contains complementary literals (a and *not* a are called complementary literals), then it is clear that no extension of A can be a stable model. Depending on the way the procedure *smodels* is used, the algorithm may start with A as an empty set, or a

```
Function expand(P,A)
repeat
  A' := A
  A := Atleast(P, A)
  A := A ⋃ { not χ | χ ∈ Atoms(P)
           and χ ∉ Atmost(P, A) }
until A = A'
return A.
```

Figure 2: Function expand(P,A)

predetermined set of literals. It adds literals into A and checks if the resulting set could lead to a stable model. If this is not the case, it will backtrack.

The *smodels'* algorithm is described in Figure 1, where P is a ground program. The function $smodels(P, A)$ returns *true* if there is a stable model of P agreeing with A. Given a partial truth assignment A, the algorithm first performs constraint propagation, which consists of two functions: $Expand(P, A)$ and $Lookahead(P, A)$. The idea is to avoid the costly calls to $smodels(P, A)$, by propagating the truth values in A to get a superset of A. Let's denote this superset as A'. Any stable model that agrees with A will agree with A'. We will give the precise definition of these functions shortly.

After the constraint propagation, the algorithm will check whether A is consistent. The function $conflict(P, A)$ returns *true* if A is inconsistent. Otherwise, it returns $false$. If A is consistent and covers all the atoms in program P, then the atom set A^+ is a stable model, and is therefore returned. If A does not cover all the atoms in P, the algorithm will choose a literal x outside of A by heuristics computed by procedure $heuristic(P, A)$, and add it to A (which means x is assigned to *true*). If the chosen literal along with A can be extended to a stable model, *true* is returned. Otherwise, the algorithm will negate the value of x, and continue.

The algorithm will be discussed in more detail in the rest of this subsection.

2.4.2 Function expand(P,A)

Figure 2 shows the detail of the function $Expand(P, A)$. The function $Expand(P, A)$ expands atom set A by using two functions: $Atleast(P, A)$ and $Atmost(P, A)$. Function $Atleast(P, A)$ returns a set of atoms A' which contains all the inevitable consequences of A. Any stable model that agrees with A must agree with atom set A'. The function $Atmost(P, A)$ returns another atom set A'', which contains all the possible consequences of A. Any literal x that is not in A'' must not be in any stable model that agrees with A, so *not* x must be in the stable model that agrees with A.

Now, we will explain $Atleast(P, A)$. Let r be a rule of program P in the following form

$$h \leftarrow a_1, ..., a_n, not\ b_1, ..., not\ b_m .$$

We say the body of a rule is $false$ w.r.t atom set A if there exists a literal $a \in A$ while its negation is in the body.

Define $min_r(A)$ to be the inevitable consequences of atom set A with respect to rule r as

$$min_r(A) = \{h \mid \{a_1, ..., a_n\} \subseteq A^+, \{b_1, ..., b_m\} \subseteq A^-)$$

It is based on the property of the stable model semantics that says for a rule, once the body is in a stable model, so is the head.

Define $max_r(A)$ to be the possible consequences of atom set A w.r.t rule r as

$$max_r(A) = \{h \mid \{a_1, ..., a_n\} \bigcap A^- = \emptyset,\ \{b_1, ..., b_m\} \bigcap A^+ = \emptyset\}$$

That is, if the body of a rule is not $false$ with respect to A, the head of the rule could be a possible consequence of set A.

Example 2.4 Let atom set $A = \{b, e, not\ t\}$ and program P be

$$\begin{array}{lll} a \leftarrow b, not\ c. & (r_1) \\ s \leftarrow b, not\ e. & (r_2) \\ d \leftarrow t, not\ c. & (r_3) \end{array}$$

For rule r_1, since $b \in A$ and $c \notin A^+$, we have $a \in max_{r_1}(A)$. For the second rule r_2, since $not\ e$ is in the body while $e \in A$, $not\ e$ is $false$ w.r.t to A, thus the head s is not in $max_{r_2}(A)$. In a similar way, the head literal d in rule r_3 is not a possible consequence of A because body literal t is $false$ w.r.t A.

Given the definition of $min_r(A)$ and $max_r(A)$, the following four rules allow the propagation of A to A', without sacrificing the semantics of the stable models. The function $expand(P, A)$ will apply the following four rules [33] repeatedly in order to propagate new atoms into set A, until no new atoms could be deduced further:

1. If $r \in P$, then $A := A \bigcup min_r(A)$.

2. If there is an atom a such that for all $r \in P$, $a \notin max_r(A)$, then $A := A \bigcup \{not\ a\}$.
   This is saying that it is impossible to deduce a from A. It includes two cases: One is that, for any $r \in P$ and a as its head, the body is $false$ with respect to A. Another case is that a does not appear in the head of any rule in program P. Since $max_r(A)$ only collects head literals into A, if a literal never appears as a head, $max_r(A)$ will never contain the literal.

3. If an atom $a \in A$, there is only one $r \in P$ for which $a \in max_r(A)$, and there exists a literal x such that $a \notin max_r(A \bigcup \{x\})$, then $A := A \bigcup \{not\ x\}$.
   This is saying that since $a \in A$, and a appears as the head literal in some rule, then there must exist at least one rule with a as head where all the body literals are in A in order to support $a \in A$. If there is only $r \in P$ which is

possible to support $a \in A$, then it must be the only rule to support $a \in A$. If there exists a literal x such that $a \notin max_r(A \bigcup \{x\})$, that means the body is $false$ once x is added into A and this rule will not support $a \in A$. While based on the stable model semantics, $not\ x$ should be included in A in order to support $a \in A$.

4. If $not\ a \in A$, and there exists a literal x such that for some $r \in P$, $a \in min_r(A \bigcup \{x\})$, then $A := A \bigcup \{not\ x\}$.
This represents the case that once the head is $false$, the body must be $false$.

Example 2.5 Program P is

$$\begin{array}{lll} h \leftarrow b, not\ e. & (r_1) \\ d \leftarrow not\ h. & (r_2) \\ a \leftarrow not\ b. & (r_3) \\ a \leftarrow d. & (r_4) \end{array}$$

Given $A = \{d\}$, we compute $Atleast(P, A)$ as follows:

1. Applying rule 1 to r_4, we get $min_{r_4}(A) = \{a\}$, then we get $A = \{d, a\}$;
2. Applying rule 2, since atom e does not appear in the head of any rule in P, we get $A = \{d,\ a,\ not\ e\}$;
3. Applying rule 3, since $d \in A$, and r_2 is the only rule with d as head, $not\ h$ will be added to A. Similarly, $not\ b$ will be added to A. Now we get $A = \{d,\ a,\ not\ e,\ not\ h,\ not\ b\}$;
4. Since no new atoms could be deduced, stop.

Now we will explain function $Atmost(P, A)$. Before defining $Atmost(P, A)$, we need to explain a proposition:

Define function f_r as

$$f_r(S, C) = \{h |\ h \leftarrow a_1, ..., a_n, not\ b_1, ..., not\ b_m \in P, \\ a_i \in C\ for\ 1 \leq i \leq n\ and\ b_j \notin S\ for\ 1 \leq j \leq m\}$$

where S and C are set of atoms.

Define

$$P_A = \{h \leftarrow l_1, ..., l_k \in P \mid not(l_i) \notin A\}.$$

Given a logic program P and a set of literals B, positive program $R(P, B)$ is defined as

$$R(P, B) = \{h \leftarrow a_1, ..., a_n \mid h \leftarrow a_1, ..., a_n, not\ b_1, ..., not\ b_m \in P \\ and\ b_i \in B^-,\ for\ i = 1, ..., m\}$$

The *deductive closure* of P and B is denoted by $Dcl(P, B)$, which is the set of atoms that contains B^+ and the minimal model of $R(P, B)$. Given the above definition, the following proposition holds:

Proposition 2.1 [32] If the stable model M of program P agrees with a set of literals A, then M agrees with

$$A \bigcup \{not\ x \mid x \in Atoms(P)\ and\ x \notin Dcl(P_A, not\ (NAnt(P)))\ \}.$$

The definition of $Atmost(P, A)$ is based on the above proposition.

Definition 2.3 [33] $Atmost(P, A)$ is the least fixed point of f', which is defined as

$$f'(B) = \bigcup_{r \in P} f_r(A^+, B - A^-) - A^-$$

where B is a set of atoms.

To compute $Atmost(P, A)$, start with B as an empty set. For every $r \in P$ that is in the form of

$$h \leftarrow not\ b_1, ..., not\ b_m.$$

h is added to B if $h \notin A^-$ and for all b_i, $i = 1, ..., m$, $b_i \notin A^+$. Then, in the following round, given B, for every $r \in P$ that is in the form of

$$h \leftarrow a_1, ..., a_n, not\ b_1, ..., not\ b_m.$$

h is added to B if $h \notin A^-$, for all b_i, $i = 1, ..., m$, $b_i \notin A^+$ and for all a_j, $j = 1, ..., n$, $a_j \in B$ but $not\ a_j \notin A$.

Example 2.6 Let program P be

$e \leftarrow not\ b.$ (r_1)
$d \leftarrow not\ h.$ (r_2)
$a \leftarrow e.$ (r_3)

Given $A = \{d\}$, $Atmost(P, \{d\}) = \{d, e, a\}$ is computed as follows:
Starting with $B = \phi$:

$f'(\phi) = f_{r_1}(\{d\}, \phi) = \{e\}$,
$f'(\phi) = f_{r_2}(\{d\}, \phi) = \{d\}$,
$f'(\phi) = f_{r_3}(\{d\}, \phi) = \phi$.
$B = \{e, d\} - A^- = \{e, d\}$
$f'(B) = f_{r_1}(\{d\}, B) = \{e\}$,
$f'(B) = f_{r_2}(\{d\}, B) = \{d\}$,
$f(B) = f_{r_3}(\{d\}, B) = \{a\}$.
$B = \{e, d\} - A^- = \{e, d, a\}$
$f'(B) = f_{r_1}(\{d\}, B) = \{e\}$,
$f'(B) = f_{r_2}(\{d\}, B) = \{d\}$,
$f(B) = f_{r_3}(\{d\}, B) = \{a\}$.
$B = \{e, d\} - A^- = \{e, d, a\}$.

Stop.

```
Function lookahead(P, A)
 repeat
  A' := A
  A := lookahead_once(P, A)
 until A = A'
return A.

Function lookahead_once(P, A)
  B := Atoms(P) - Atoms(A)
  B := B ⋃ not (B)
  while B ≠ φ do
    take any literal χ ∈ B
    A' = expand(P, A ⋃ {χ})
    B := B - A'
    if conflict(P, A') then
      return expand(P, A ⋃ {not (χ)})
    end if
 end while
return A.
```

Figure 3: Function lookahead(P,A)

2.4.3 Function lookahead(P,A)

Lookahead is based on the idea of testing every possible choice before committing to one. In the case of *smodels*, this means that we can avoid choosing a literal that leads to a conflict. That is, if the stable model M agrees with the set of literals B but does not agree with $B \bigcup \{x\}$, it holds that M must agree with $B \bigcup \{not\ x\}$. Thus lookahead immediately provides a stronger pruning method: if $A' = Expand(P, A \bigcup \{x\})$ has conflicts, add *not* x to A. Lookahead has been shown to be the most effective cost-saving method used in *smodels* [30].

The function $Lookahead(P, A)$ is given in Figure 3. $Lookahead(P, A)$ will call function *lookahead_once*(P,A), which picks up a literal from outside of A, say x, and expands A with x. If $A' = Expand(P, A \bigcup \{x\})$ and $conflict(P, A')$ returns *true*, then $Expand(P, A \bigcup \{not\ x\})$ is called. The function *lookahead_once*(P,A) will try all the literals and their negations outside of A.

2.4.4 Function heuristic(P,A)

The heuristics can in principle return any atom in $Atoms(P)$ for a program P. Since $smodels(P, A)$ calls itself twice for every choice point that the heuristics finds, the algorithm explores in the worst case a search space of size $2^{|Atoms(P)|}$. To restrict the choice points to a smaller set will substantially prune the search space. *Smodels* applies this by restricting all choice points to the atoms that appear as not-atoms in the bodies of the rules, or in the heads of the choice rules [33].

2.5 Extended rules in logic programming

In order to express some applications more conveniently, logic programs have been extended with new constructs, including *choice rules*, *cardinality constraint rules* and *conditional literals*. It has been shown that all the constructs can be translated into basic rules of the form

$$h \leftarrow a_1, ..., a_n, not\, b_1, ..., not\, b_m.$$

along with primitive cardinality rules. In this subsection, we will describe the syntax of these extended constructs and their translations to the basic rules.

Choice rule [28] is of the form

$$\{h_1, \ldots, h_m\} \leftarrow body.$$

If the body of a choice rule is satisfied, any subset of the atoms in the head could be included in a stable model. It is said that the *body* gives the head literals a reason to be in the model, but it does not force them to be in it [36].

The above choice rule is equivalent to the following normal rules: For each literal h_i in the head, we have:

$$\begin{aligned} h_i &\leftarrow not\ h_i', body. \\ h_i' &\leftarrow not\ h_i, body. \end{aligned}$$

where h_i are new literals.

Cardinality constraint [28] is of the form

$$L\{a_1, ..., a_n, not\, b_1, ..., not\, b_m\}U$$

where L and U are two integers giving the *lower* and *upper bounds* of the constraint, respectively. The constraint is satisfied by a model if the cardinality of the subset of the literals satisfied by the model is between the integers L and U, inclusively. Either of the bounds may be omitted, in which case a missing lower bound is to be taken as $-\infty$ and an upper bound as ∞. A *cardinality constraint rule* is a rule with cardinality constraints in the body or head.

A *primitive cardinality constraint rule* is of the form

$$h \leftarrow L\{l_1, ..., l_n\}.$$

where h is an atom and l_i are literals. Let $|A_S|$ denote the cardinality of literal set A_S which contains all the literals in A that is satisfied by model S. Based on the stable model semantics, if $|\{l_1, ..., l_n\}_S| \geq L$, then h must be satisfied by S.

A *cardinality constraint rule* [28] can be represented in the following form

$$L\{h_1, ..., h_k\}U \leftarrow B_1, ..., B_n.$$

where h_i is an atom and B_i is a cardinality constraint or literal.

To illustrate the translation of cardinality rules to basic rules, we take the simple rule

$$L_h\{h_1, ..., h_k\}U_h \leftarrow L_b\{b_1, ..., b_k\}U_b,\ b.$$

as an example, where b_i is a literal and b represents a body literal. The cardinality constraint $L_b\{b_1, ..., b_k\}U_b$ is translated into

$$new_{b1} \leftarrow L_b\{b_1, ..., b_k\}.$$
$$new_{b2} \leftarrow (U_b + 1)\{b_1, ..., b_k\}.$$

where new_{b1} and new_{b2} are new atoms. The head $L_h\{h_1, ..., h_k\}U_h$ is translated into the rules

$$new_{h1} \leftarrow L_h\{h_1, ..., h_k\}.$$
$$new_{h2} \leftarrow (U_h + 1)\{h_1, ..., h_k\}.$$

where new_{h1} and new_{h2} are new atoms. Then, the cardinality rule is encoded as

$$new \leftarrow new_{b1}, not\ new_{b2}, b.$$
$$\{h_1, ..., h_k\} \leftarrow new.$$
$$\leftarrow new,\ not\ new_{h1}.$$
$$\leftarrow new,\ new_{h2}.$$

Finally, in order to write down sets of literals needed in constraint expressions compactly, *conditional literals* of the form

$$l : d$$

are introduced into logic programs where l is a literal and the conditional part d is a domain predicate. It is a shorthand for expressing a set of ground literals. For example,

$$1\{setColor(v, C)\ :\ color(C)\}1$$

is equivalent to

$$1\{setColor(v, red), setColor(v, green), setColor(v, blue)\}1$$

when the domain predicate $color(C)$ corresponds to a set of facts

$$color(red).$$
$$color(green).$$
$$color(blue).$$

Although the extended constructs extend normal logic programming considerably, the computational complexity remains to be NP-complete [29].

3 Encoding of Planning in Logic Programming

In this chapter, we discuss how to encode planning problems in logic programming. The idea is that the action predicates that are *true* in a stable model of the logic program correspond to a valid plan.

First, in Section 3.1, we relate planning to the action theory. According to the action theory, any planning problem consists of three parts: initial state, goal state and an action system. In the following three sections (Section 3.2, 3.3 and Section 3.4), a method to encode these three parts into logic programming is explained. In Section 3.5, a planning benchmark, the gripper problem, is taken as an example to illustrate the discussed method.

3.1 Action theory and planning

A *transition system* consists of a set of states S described by a set of fluents, a set of actions A, and a subset R of $S \times A \times S$. $r = \langle s, a, s' \rangle \in R$ means that after the execution of action $a \in A$ in state $s \in S$, one will get another state $s' \in S$ [16].

An *action theory* is a pair (D, Ψ), where D consists of propositions indicating when an action is executable, and how the state conditions change after an action takes place. We call D an *action system*. Ψ consists of propositions to describe the initial state.

Given a transition system, a *trajectory* of the transition system is a sequence, $s_0, a_0, s_1, \ldots, a_{n-1}, s_n$ where $\langle s_i, a_i, s_{i+1} \rangle$ is a relation $r \in R$. Let Δ be a formula. We then say $s_0, a_0, s_1, \ldots, a_{n-1}, s_n$ is a *trajectory* of formula Δ if Δ holds in s_n.

A planning problem can be defined using action theory.

Definition 3.1 A *planning problem* is a triple (D, Ψ, Δ) where (D, Ψ) is an action theory and Δ is the goal which holds at the goal state. A sequence of actions $a_1, a_2, ..., a_n$ is a possible plan for goal Δ if $s_0, a_0, s_1, \ldots, a_{n-1}, s_n$ (where s_0 satisfies Ψ) is a trajectory of Δ.

To represent a planning problem (D, Ψ, Δ) in logic programming, three tasks are needed:

1. representing *fluents* and *actions*,
2. representing *initial* and *goal* states, and
3. representing action system D

3.2 Representing fluents and actions in logic programming

The fluents and actions are represented by predicates in logic programs. There are state-related fluents which represent state specific information, and non-state fluents, which represent information that holds for all states. For actions and state-related fluents, we add a state parameter T to explicitly represent which state the fluents and actions are about. For example, if the fluent $at(a, table)$ is used to describe state

conditions for states 1 to 3 , the fluent $at(a, table)$ will be encoded in a logic program as

$at(a, table, 1).$
$at(a, table, 2).$
$at(a, table, 3).$

For example, if predicate $at(a, table, 3)$ is *true*, it means that $at(a, table)$ holds in state 3.

Non-state fluents can be represented by predicates without state parameters, such as $numberGripper(robot, 2)$ which means "a robot has two grippers".

3.3 Representing initial and goal states in logic programming

Given a planning domain, the initial state is represented by the state-related predicates with the state parameter being *zero*.

The goal state is represented using a predicate $goal(T)$, it is *true* means that goals are reached at state T. The following rule is designed to put together all fluents that describe the goal state:

$$goal(T) \leftarrow goal_1(T), ..., goal_n(T).$$

where $goal_i$ is a fluent that must be satisfied in the goal state.

Since it is possible that before the last state, goal conditions are reached, then from that state on, no actions need to be taken. Therefore any state after that will always satisfy the goal conditions. The following rule represents this logic:

$$goal(T+1) \leftarrow goal(T).$$

In order to say that goal conditions must be reached after applying a plan, for clarity, a new atom *goal* is introduced to construct the following rules:

$goal \leftarrow goal(T).$
$\leftarrow not\ goal.$

3.4 Representing action systems in logic programming

Now, we will explain how to encode the action system. In this thesis, concurrency is allowed in the encodings. That is, parallel actions are allowed to happen in the same state. We can also specify nonparallel actions.

To completely encode an action system for planning, four groups of rules are needed:

1. **Action Choice**: Which actions should be chosen.
2. **Affected Objects**: What objects are affected by an action.
3. **Effects**: If affected, what are the effects on the affected objects.

4. **Frame axioms**: If not affected by any action at a state, the fluents that hold at the current state remain to hold at the next state

We now discuss these rules in more detail.

Action Choice

A rule describing under what preconditions an action may take place is called an *action rule*. An *action rule* can be encoded as a rule, with the action as head and preconditions of the action as body:

$$action(Obj_1, ..., Obj_n, T) \leftarrow preconditions.$$

The predicate $action(Obj_1, ..., Obj_n, T)$ represents the action occurring at state T, and Obj_i $(1 \leq i \leq n)$ are objects related to the action. Based on stable model semantics, once the preconditions are included in a stable model, the action must be added to the model.

By the above rule alone, conflicting actions may arise. Two actions are *conflicting* if they have the same preconditions but their effects are in conflict; therefore they cannot occur at the same state. For example, a block a could be moved to block b or c, if the three blocks are clear. But block a can only be moved to either of b or c at any one state. Conflicting actions present multiple choices for the planner to choose from.

There are two ways to exclude conflicting actions. One is to use a cyclic structure as given below:

$$\begin{aligned} &action(Obj_1, ..., Obj_n, T) \leftarrow \\ &\qquad preconditions, \\ &\qquad not\ blockedAction(Obj_1, ..., Obj_n, T). \qquad \{1\} \end{aligned}$$

$$\begin{aligned} &blockedAction(Obj_1, ..., Obj_n, T) \leftarrow \\ &\qquad not\ action(Obj_1, ..., Obj_n, T). \qquad \{2\} \end{aligned}$$

$$\begin{aligned} &blockedAction(Obj_1, ..., Obj_n, T) \leftarrow \\ &\qquad conflictingAction(Obj_1, ..., Obj_n, T). \qquad \{3\} \end{aligned}$$

As shown in rules {1} and {2}, $action(Obj_1, ..., Obj_n, T)$ takes place if all its preconditions are satisfied and $blockedAction(Obj_1, ..., Obj_n, T)$ is $false$, meaning there is nothing to block the action. $blockedAction(Obj_1, ..., Obj_n, T)$ is *true* if the action $action(Obj_1, ..., Obj_n, T)$ does not occur at state T. This cyclic structure presents the possibility that $action(Obj_1, ..., Obj_n, T)$ may occur, or may not. If a conflicting action of $action(Obj_1, ..., Obj_n, T)$ takes place at state T, $blockedAction(Obj_1, ..., Obj_n, T)$ is *true*. This is expressed by rule {3}.

This cyclic structure is important to express that conflicting actions should not happen at the same state. Any stable model that contains $action(Obj_1, ..., Obj_n, T)$ cannot contain $blockedAction(Obj_1, ..., Obj_n, T)$, and vice versa.

Another way to encode action choice is to use "choice rules + constraints" in the following form:

$$\{action(Obj_1, ..., Obj_n, T)\} \leftarrow preconditions. \quad \{4\}$$

$$\leftarrow action(Obj_1, ..., Obj_n, T), conflictingAction(Obj_1, ..., Obj_m, T). \quad \{5\}$$

Rule {4} is a choice rule, which means that if all the preconditions are satisfied, the action $action(Obj_1, ..., Obj_n, T)$ may take place, or may not. However, using choice rule alone, all the actions may take place when their preconditions are satisfied, including conflicting actions. In order to prevent conflicting actions, we construct constraints in form of {5}. Based on the stable model semantics, rule {5} expresses the logic that $action(Obj_1, ..., Obj_n, T)$ and $conflictingAction(Obj_1, ..., Obj_m, T)$ cannot be *true* in the same stable model. Thus, the rule can explicitly prevent $action(Obj_1, ..., Obj_n, T)$ and $conflictingAction(Obj_1, ..., Obj_m, T)$ from happening at the same state.

Since conflicting actions lead to conflicting effects, inference rules to prevent conflicting effects can implicitly prevent conflicting actions. For example, in the blocks world problem, $on(a, b, T)$ and $on(a, c, T)$ are effects of conflicting actions $move(a, b, T-1)$ and $move(a, c, T-1)$. So instead of rule {5} and {4}, we can construct rule {6} and {4} to encode action choice.

$$\{action(Obj_1, ..., Obj_n, T)\} \leftarrow preconditions. \quad \{4\}$$

$$\leftarrow property(Obj, T), conflictingProperty(Obj, T). \quad \{6\}$$

Using "choice rules + constraints" to encode action choices is preferable to using cyclic structure. First, the former does not need predicate $blockedAction(Obj_1, ..., Obj_n, T)$, therefore the encoding is more compact. Secondly, as shown in our experiments, the former is more efficient in the Smodels system. Remember that Smodels restricts its choice points to atoms that appear as not-atoms in the bodies of rules and atoms in the heads of choice rules. Accordingly, these two ways to construct *action rule* result in a different search space. The atoms from predicates $action(Obj_1, ..., Obj_n, T)$ and $blockedAction(Obj_1, ..., Obj_n, T)$ are choice points in the latter encoding while in the former encoding, only atoms from $action(Obj_1, ..., Obj_n, T)$ are choice points. The number of choice points is reduced.

Affected Objects

The state properties may be changed by actions. To model the effects of an action in logic programs, first we need to identify the objects that are affected by the action.

An object is *affected* by an action if the properties of the object change after the action takes place. To identify affected objects, the following rule is constructed to express that object Obj is affected by action *action* at state T.

$$affected(Obj, T) \leftarrow action(T). \quad \{7\}$$

Effects

The rules of the following form are constructed to express the effects of actions:

$$property_2(Obj, T+1) \leftarrow action(T), property_1(Obj, T). \quad \{8\}$$

where Obj is an affected object by action $action(T)$. This rule says that the property of object Obj is changed from $property_1(Obj, T)$ to $property_2(Obj, T+1)$ because action $action$ takes place at state T.

If the action itself can determine the property of the affected object after the action, $property_1(Obj, T)$ can be omitted in $\{8\}$.

Frame Axioms

In order to formulate that a property of an object that holds at the current state will continue to hold at the next state, since no actions affect the property at the current state, *frame axioms* need to be encoded. The following construct can express this:

$$property(Obj, T+1) \leftarrow not\ affected(Obj, T), property(Obj, T). \quad \{9\}$$

For a logic program constructed using the above pattern for a planning problem, it holds that the program has a stable model, if and only if there is a sequence of actions that can be executed to change the initial state to the goal state in given steps. The atoms in a stable model of the program include atoms that describe state properties and actions executed. The atoms with state parameter t that are *true* in a stable model provide complete information about state t. The actions that are *true* in a stable model make up a valid plan for the underlying planning domain.

In the next section, we will take the gripper problem as an example in order to show how to apply the pattern just described.

3.5 Encoding of the gripper problem

Figure 4 shows an example of the gripper problem [24]. The goal of this problem is to transport all the balls from one room to the other room. To accomplish this, the robot is allowed to move from one room to the other, and use its two grippers to pick up, and put down balls. Each gripper can hold one ball at a time. A solution to this example is shown in Table 1. Notice that parallel actions are allowed: two grippers could pick up or put down balls at the same state.

The objects in the gripper problem are *balls*, two *grippers*, and a *robot*. We use two predicates to describe properties for *balls*:

1. $at(A, R, T)$: ball A is at room R at state T
2. $on(A, G, T)$: ball A is on gripper G at state T

Two predicates are used to specify properties of grippers:

1. $free(G, T)$: gripper G is free at state T, not holding any ball

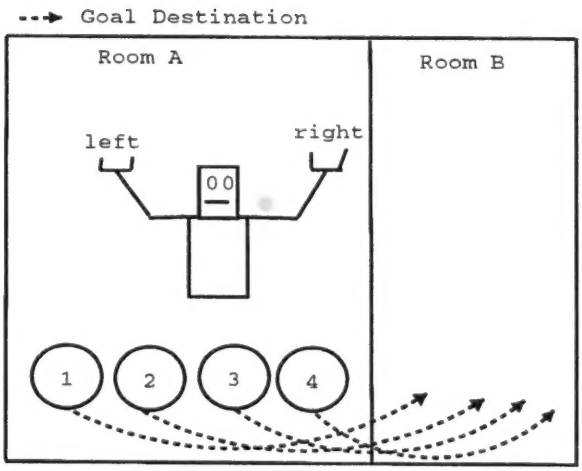


Figure 4: An example for the gripper problem

state	action
0	Left gripper picks up $ball_1$,Right gripper picks up $ball_2$
1	*Robot* moves to *room B*
2	Left gripper puts down $ball_1$, Right gripper puts down $ball_2$
3	*Robot* moves to *room A*
4	Left gripper picks up $ball_3$,Right gripper picks up $ball_4$
5	*Robot* moves to *room B*
6	Left gripper puts down $ball_3$, Right gripper puts down $ball_4$

Table 1: An optimal plan for the gripper problem instance in Figure 4

2. *holding*(G,T): gripper G is holding a ball at state T

The predicate *at*(*robot*, R,T) is used to describe the properties of the *robot*, which reads " *robot* is at room R at state T ."

We first describe the encoding for the initial and goal states of the gripper domain using the example shown in Figure 4. Then, we give the encoding of the action system and explain the encoding in detail.

3.5.1 Encodings of initial and goal states

The initial and goal conditions for the gripper problem with four balls as in Figure 4 are presented as follows:

```
%facts
room(room_A). Room(room_B).
gripper(left). gripper(right).
ball(a). ball(b). ball(c). ball(d).

%initial state
at(robot,room_A,0).
free(left,0).
free(right,0).
at(a,room_A,0).
at(b,room_A,0).
at(c,room_A,0).
at(d,room_A,0).

%goal state
goal(T)←
    state(T),
    at(a,room_B,T),
    at(b,room_B,T),
    at(c,room_B,T),
    at(d,room_B,T).

goal(T+1)← goal(T).
goal← goal(T).
← not goal.
```

3.5.2 Encoding of the action system

There are three actions in this problem domain and they are listed in the following table.

pickup(G,A,T)	gripper G picks up ball A at state T
putdown(G,A,T)	gripper G puts down ball A at state T
move(*robot*, R_1,R_2,T)	*robot* moves from room R_1 to room R_2 at state T

The preconditions of action *pickup*(G,A,T) are as follows:

1. gripper G is free (not holding anything);
2. *robot* and ball A are in the same room;
3. *robot* is not affected (*robot* is not moving);
4. goal conditions are not reached.

The preconditions of action $putdown(G, A, T)$ are:

1. robot is not affected at state T (not moving);
2. ball A is on gripper G;
3. goal conditions are not reached.

The preconditions for action $move(robot, R1, R2, T)$ are:

1. robot is in room R_1;
2. room R_2 is not the same room as R_1;
3. goal conditions are not reached.

Program 3.1 is a complete encoding of the action system for the gripper problem.

Program 3.1 The encoding of the action system for the gripper problem

```
%written by Xiumei Jia

% specify the domain of variables:
% state(T),room(R1),room(R),grip(G),grip(G1)
% ball(A),ball(B),ball(C)

% action rules:

% r1
{pickup(G,A,T)}←
   free(G,T),
   at(robot,R,T),
   at(A,R,T),
   not affected(robot,T),
   not goal(T).
% r2
{putdown(G,A,T)}←
   at(robot,R,T),
   on(A,G,T),
   not affected(robot,T),
   not goal(T).
% r3
```

```
{move(robot,R1,R,T)}←
      at(robot,R1,T),
      R1 ≠ R,
      not goal(T).

%  Constraints:

% r4,r5 and r6
← 2{at(A,R,T):room(R),on(A,G,T):gripper(G)},ball(A),state(T).
← 2{on(A,G,T):ball(A)}, gripper(G),state(T).
← 2{at(robot,R,T):room(R)},state(T).

% Identify affected objects:

%for pickup: r7 and r8
affected(A,T)←
    pickup(G,A,T).
affected(G,T)←
    pickup(G,A,T).
% for putdown: r9 and r10
affected(A,T)←
    putdown(G,A,T).
affected(G,T)←
    putdown(G,A,T).
%for move: r11
affected(robot,T)←
    move(robot,R1,R,T).

%effects

% pickup: r12 and r13
on(A,G,T+1)←
    pickup(G,A,T).
holding(G,T+1)←
    pickup(G,A,T).

%putdown: r14 and r15
free(G,T+1)←
   putdown(G,A,T).
at(A,R,T+1)←
   at(robot,R,T),
   putdown(G,A,T).

%move: r16
at(robot,R2,T+1)←
```

```
        move(robot,R1,R2,T),
        R1 ≠ R2.

% frame axioms

at(A,R1,T+1)←
    at(A,R1,T),
    not affected(A,T).

on(A,G,T+1)←
   on(A,G,T),
   not affected(A,T).

free(G,T+1)←
    not affected(G,T),
    free(G,T).

holding(G,T+1)←
    not affected(G,T),
    holding(G,T).

at(robot,R,T+1)←
   at(robot,R,T),
   not affected(robot,T).
```

Now, we explain the above program in detail.

Action Choice

The action choices are encoded using "choice rules + constraints." For each action, an action rule is encoded using a choice rule, then constraints are constructed to prevent conflicting actions. Rules r_0, r_1 and r_3 are the action rules for actions *pickup*, *putdown* and *move* respectively.

Rules r_3, r_4 and r_5 are constraints to rule out conflicting effects for all the objects, thus preventing conflicting actions implicitly. Rule r_3 states that "ball A cannot be in more than two places at a state." Rule r_4 states that "a gripper cannot hold more than one ball at one state." These two constraints are used to prevent conflicting properties for *ball* and *gripper*, caused by conflicting actions of *pickup* and *putdown*. Constraint r_5 expresses that the *robot* cannot stay in more than one room at the same state. This constraint will prevent conflicting properties for *robot* caused by actions *move*, *pickup* and *putdown*.

Affected Objects

After action $pickup(G, A, T)$ takes place, ball A will be on gripper G instead of staying on floor, gripper G will hold a ball, and is not free anymore, while the *robot* will stay in the same place. Therefore, action $pickup(G, A, T)$ will affect ball A and gripper G. Rules r_6 and r_7 are constructed to identify ball A and gripper G as the

affected object. Similarly, Rules r_8 and r_9 are constructed to identify ball A and gripper G as the affected object of action $putdown(G, A, T)$.

Action *move* will not affect any ball since, if before moving, a ball is on a gripper, then after moving, the ball will still stay on the gripper. Also, if a ball is at room R, then before and after moving, the ball will still stay where it is. The action $move(robot, R_1, R_2, T)$ will not affect any gripper since, during the moving, a gripper cannot do *pickup* or *putdown*. Rule r_{10} is used to identify that only the *robot* is affected by action *move*.

Effects

As we have mentioned, after action $pickup(G, A, T)$ takes place, ball A will be on gripper G while gripper G will hold a ball. We construct rules r_{11} and r_{12} respectively to describe the effects of $pickup(G, A, T)$ to them. After action $putdown(G, A, T)$ takes place, ball A will be in the room where the *robot* is and gripper G will be free. Rules r_{13} and r_{14} describe these effects of $putdown(G, A, T)$. The effect of $move(robot, R_1, R_2, T)$ is that the *robot* is in room R_2 at state $T+1$. This is expressed by rule r_{15}.

Frame Axioms

For all the objects that are not affected by any actions at state T, inference rules are constructed to represent that their properties continue to hold at the next state, as described in rules r_{16} to r_{20}.

4 Previous Work on Domain Dependent Knowledge

4.1 CPlan

CPlan [4] is a planner based on constraint satisfaction problems (CSPs). A CSP problem consists of:

1. a set of variables $X = \{x_1, ..., x_n\}$,
2. for each variable x_i, a finite set D_i of possible values (its domain),
3. a set of constraints restricting the values that the variables can simultaneously take.

A solution to a CSP is an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied.

A CSP is often solved using the backtracking paradigm. Backtracking attempts to incrementally extend a partial solution toward a complete solution, by repeatedly choosing a value for a variable. To reduce search space, consistency techniques are applied during the backtracking search, after a variable is assigned a value. Consistency technique is to discover what values are not possible so they can be safely removed from the domain temporarily (they will be recovered upon backtracking).

Node consistency and *arc consistency* are the most common consistency techniques used. *Node consistency* applies only to constraints with one unassigned variable. It checks what values for this variable are not possible, should therefore be removed. *Arc consistency* applies to a constraint with two unassigned variables, say X_1 and X_2. If, for a value of X_1, there is no value for X_2, such that the constraint can be satisfied, then this value for X_1 can be removed. This is done repeatedly until no domain reduction is possible.

Many other techniques have been developed to reduce the size of the search space including adding redundant variables and redundant constraints to the CSP model .

CPlan takes advantage of CSP and treats the planning problems as CSP. CPlan models each state by a collection of variables and constraints which enforce valid transitions between states. There are no variables that explicitly model actions. The actions are extracted from the changes of the variables. For the logistics problem which involves moving packages to different locations of different cities by trucks and planes, CPlan has the following variables for each state: S_t, $C_{i,t}$, $T_{j,t}$, $P_{k,t}$ where i ranges over the number of packages, j and k range over the number of trucks and planes, respectively, and t ranges over the number of steps in the plan. The domains of the package variables $C_{i,t}$ are locations, trucks, and planes. Assigning a package variable a location means the package is at that location and assigning package variable a truck means the package is in that truck. Similarly, the domains of trucks and planes are locations.

In CPlan, the following constraints can be used to represent a planning problem:

Action constraints:
This kind of constraint models the effects of actions.

State constraints:

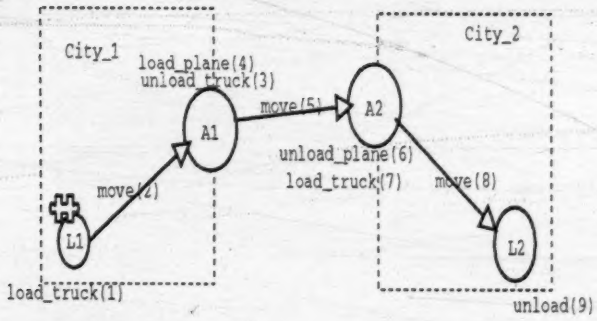


Figure 5: An example for distance constraint in the logistics problem

This kind of constraint enforces how variables within a state must be consistent.

Domain constraints:

This kind of constraint enforces restrictions on the original domains of the variables. For example, in the logistics domain, a package which is to be picked up and delivered within the same city can have its domain restricted to locations and trucks within that city.

Capacity constraints:

This kind of constraint enforces bounds on resources. For example, in the gripper problem, one gripper could hold one ball at a time; a truck in the logistic problem can carry unlimited number of packages.

Distance constraints:

These kinds of constraints are upper and lower bound constraints on the number of steps needed for a variable to change from one value to another. For example, a lower bound on the number of steps to get a package from a non-airport location in one city to a non-airport location in another city is nine steps (as it needs to be loaded and unload from two trucks and one plane) as shown in Figure 5.

CPlan searches plans by setting the length of the plan as the lower bound. If there is no solution, CPlan will try in the next round by incrementing the length of the plan. When a planning instance has no plan, CPlan will not run for forever. It computes an upper bound, say *ULength*. If CPlan cannot find a valid plan with length *ULength*, then it will stop and conclude that there is no solution for this instance.

Symmetric constraints:

This kind of constraint consists of constraints which break symmetries on the values that variables can be assigned. For example, in the logistics domain, given two package variables, the planes in their domains are often symmetric and if there is a solution (or no solution) with a particular assignment of planes to packages, there is another solution (or no solution) with the planes swapped. With distance constraints, It is claimed that these constraints were found to be the most important for reducing the search space in the domains.

Action choice constraints:

This kind of constraint enforces constraints on which actions can be performed in each state. Part of the explosion in the search space in planning is because a

sequence of actions starting from some state can be permuted and still result in the same end state. For example, for the logistics problem, suppose there are two packages at an airport. A plane can either pick up both at once, or pick up one now and another later. All of these will produce the same effects, and a constraint is added which forbids all but one of the action sequences.

Constraints can be classified as redundant or non-redundant. A constraint is redundant if its removal from the CSP does not change the set of solutions. Of all these constraints, action constraint, state constraint, capacity constraint and domain constraint are non-redundant constraints. These non-redundant constraints are necessary to define the planning domain in CSP. Others are redundant constraints which are used to introduce domain dependent knowledge into the search.

In CPlan, part of the modeling task is to specify what kind of propagation is desired for each constraint: whether a constraint should be forward checked, or arc consistency checked, or needs other techniques. This strategy provides a chance to apply best technique to each constraint according to its properties, thus taking advantage of these techniques. Since each individual problem needs a CSP model to represent, the disadvantage of this strategy is that the user has to know the search algorithm and domain dependent knowledge at the same time, and he/she must have knowledge about which specific search method is best for a specific constraint. This disadvantage prevents CPlan becoming widely used.

4.2 TLPlan

TLPlan [1] is another planning system utilizing domain dependent knowledge. It represents domain dependent knowledge in a temporal logic and utilizes it to control a forward-chaining planner.

In the TLPlan system, planning domains are encoded using STRIPS. For example, the gripper problem with a one-hand robot is encoded using the following STRIPS operators:

Operator	Preconditions and Deletes	Adds
pickup(A)	in(A,R),handFree,in(robot,R)	holding(A)
putdown(A)	holding(A)	in(A,R),handFree,in(robot,R)
move(R_1,R_2)	in(robot,R_1)	in(robot,R_2)

Domain dependent knowledge cannot be expressed by STRIPS. This is because most domain dependent knowledge involves more than one state, while STRIPS only involves one state. In order to use domain dependent knowledge, a specific language using first order temporal logic [3] has been designed for TLPlan, where some later states can be referred.

The language starts with a standard first-order language, ζ, containing a collection of constants, functions, and predicate symbols, along with a collection of variables. The propositional constants *true* and *false* are also included in the language and treated as atomic formulas. TLPlan adds the following temporal modalities to ζ: $\bigcup$ (until), $\Box$(always), $\diamond$(eventually), and $\circ$(next).

The standard formula formation rules are augmented by the following rule:

If $f_1$and f_2 are formulas then so are $f_1 \bigcup f_2$, $\natural f_1$, $\diamond f_1$, and $\circ f_1$.

The extension of ζ to include these temporal modalities is called $\zeta\Gamma$.

$\zeta\Gamma$ is interpreted over sequences of states, and the temporal modalities are used to assert properties of these sequences. The temporal modalities could be interpreted as follows:

1. $\circ f_1$ means that f_1 is true at the next state
2. $\natural f_1$ means that f_1 is true at the current state and at all later states
3. $\diamond f_1$ means that f_1 either holds now or at some later state
4. $f_1 \bigcup f_2$ means that either now or at some later state f_2 is true and until then f_1 becomes true.

Here are some examples of what can be expressed in $\zeta\Gamma$. Given $M = \{s_0, s_1, ...\}$ as a sequence of states:

1. $M \models \circ \circ on(A, B)$ means that $on(A, B)$ must be true in the third state s_2.
2. $M \models \natural \neg holding(A)$ means that $holding(A)$ will never happen in all the states
3. $M \models \natural (on(B, C) \Rightarrow (on(B, C) \bigcup on(A, B)))$ means that whenever $on(B, C)$ is *true* in a state, it remains *true* until $on(A, B)$ becomes true. That is, $on(B, C)$ is preserved until we achieve $on(A, B)$ for all the states.

In TLPlan, an algorithm is designed to first interpret language $\zeta\Gamma$, then to incorporate the interpretation to its forward-chaining planning algorithm. The algorithm starts with the initial state, followed by the first state, and so on. Every state is labeled with an $\zeta\Gamma$ formula. The formula controls the choice of actions in the following way:
Given a partial plan which consists of the actions taken in previous states, if adding an action to the partial plan makes the formula $false$, then choose another action. Otherwise, continue to next state.

The efficiency is improved by applying domain dependent knowledge encoded in $\zeta\Gamma$, but TLPlan cannot guarantee the optimal plan. The plans generated by TLPlan are often twice the length of the optimal plans [1].

5 Domain Dependent Knowledge in Logic Programming

Almost all the planning domains have domain dependent knowledge. How to extract the knowledge out of the domain structures is of important research interest. The classification described in Section 5.1 is motivated to direct users to find domain dependent knowledge.

If the domain dependent knowledge does not rule out any optimal plan for the planning domains, we call the knowledge *optimality* knowledge (adapted from [20]). All the domain dependent knowledge discussed in this thesis is optimality knowledge.

Once domain dependent knowledge is found, a formalization and a standard encoding are provided for each class. We hope that the formalization and encodings could provide a guide to automatically incorporating the knowledge into logic programs which encode the domain independent knowledge of planning problems, thus improving the search efficiency of planning. Each of Sections 5.2 to 5.6 discusses in detail one class of domain dependent knowledge. Some examples are provided to illustrate the encodings.

5.1 Classification of domain dependent knowledge

In this section, a classification is proposed. The classification is based on the nature of the knowledge, not on the properties of underlying logic programming solvers. The aim of the classification is to summarize the properties for different classes of domain knowledge, so that if there is a new planning domain, the users can investigate domain knowledge according to the classification, and extract the right knowledge quickly.

Domain dependent knowledge can be classified into the following five categories.

End state knowledge

End state knowledge (ESK) is the domain information about the initial state and goal state. It extracts static information about these end states.

State relationship knowledge

State relationship knowledge (SRK) is the domain dependent knowledge about relationships among two or more states.

There are two subclasses of state relationship knowledge. One consists of relationships among state conditions and actions. Under certain state conditions, even if the preconditions of an action are satisfied, the action should not be taken since executing the action will introduce *redundant actions*. An action A in a plan M is *redundant* if the removal of A from M still results in a valid plan. This type of knowledge tightens the preconditions for some actions.

For example, "move block a from table to table" falls into this category in the blocks world problem. The goal of the blocks world problem is to move a number of blocks from the initial configuration to the goal configuration. A block can only be moved to another block or a table at one state. The preconditions of action $move(a, table)$ are "block a is clear, the table is clear, and goal state has not been reached". But even if the preconditions of this action are satisfied, the action should not take place

since it does not change the properties of block a. Removing action $move(a, table)$ under the condition "block a is on table" will result in a valid plan.

Another subclass concerns the relationships among properties of different states. Prohibiting certain relationships in related states can implicitly prune redundant actions. For example, in exploring the gripper problem, the knowledge "return a ball back to its original room" is captured by a relationship which relates the properties of three states. Let A be a ball, R be a room, and T_i be a state, then the relationship can be described as a set of properties: $\{at(A, R, T_1), not\ at(A, R, T+1), at(A, R, T_2)\}$, where $T_2 > T + 1$. This relationship should be prohibited since the total effect is nothing. Therefore, all the actions that make the relationship happen are redundant.

Procedural knowledge

Procedural knowledge (PK) is the knowledge about the sequential information of the actions in final plans.

If the sequential pattern in final plans can be extracted, the information can be used as extra constraint to prune bad plans. During the search, any partial plan where actions are not in the specified sequence should be pruned.

Procedural knowledge and state relationship knowledge are different. The former focuses on the sequence of the actions while the latter focuses on the preconditions of actions.

Symmetry knowledge

Symmetry knowledge (SK) is the knowledge of checking and breaking symmetries.

Definition 5.1 Two plans M_1, M_2 are *isomorphic* if M_1 can be obtained from M_2 by exchanging an object e with another object e' in M_2. Object e and object e' are called *symmetric objects*.

For example, in a gripper domain, room R_1 has two balls (ball a, ball b) and the robot has only one gripper (we call it *gripper*), the goal is moving both balls to another room R_2. The following plans are isomorphic:
M_1:

$$pickup(gripper, a, 0), move(robot, R_1, R_2, 1), putdown(gripper, a, 2),$$
$$move(robot, R_2, R_1, 3), pickup(gripper, b, 4), move(robot, R_1, R_2, 5),$$
$$putdown(gripper, b, 6)$$

M_2 :

$$pickup(gripper, b, 0), move(robot, R_1, R_2, 1), putdown(gripper, b, 2),$$
$$move(robot, R_2, R_1, 3), pickup(gripper, a, 4), move(robot, R_1, R_2, 5),$$
$$putdown(gripper, a, 6)$$

Plan M_1 can be obtained if we exchange ball a with ball b in plan M_2, thus ball a and ball b are symmetric objects.

Many planning domains have symmetric objects. The symmetries will give rise to an (often exponential) amplification of the search space. Eliminating symmetries can therefore improve search efficiency significantly.

Distance knowledge

Distance knowledge (DK) is the knowledge concerning whether the distance from the current state to the goal state is long enough to change the properties of an object (or all objects of a type) to its (or their) properties that hold in the goal state.

A *lower bound* is computed at any state to represent the minimum steps needed to change properties of an object (or all objects of a type) that hold at the current state to properties that hold at the goal state. Suppose T is the current state, ***steps*** is the given length of plans. If the lower bound for an object is greater than $(\boldsymbol{steps} - T)$, that means the left steps are not long enough to get a plan. It also means that the current partial plan will not produce any valid plan, therefore the search should stop extending it. In this way, the search can stop an unpromising branch at an early stage.

5.2 End state knowledge

The initial state and goal state are static. These two states actually provide important facts about planning domains and while this information appears minimal, the effects can be huge.

The initial and goal properties of some objects can provide information to exclude redundant actions. Given the initial or goal properties of an object, taking certain actions on this object may be redundant, or not executing the actions may introduce redundant actions.

For example, in the gripper problem, if a ball is already in the goal room, then *pickup* will be a redundant action for this ball. However if the ball is on a gripper and the robot is not in the ball's goal room, *putdown* is a redundant action.

According to the above discussion, the information derived from ESK can be formalized using the following rules:

$$IF\ \chi\ THEN\ not\ Action \qquad (5.1)$$

where χ represents the initial or goal properties of an object which holds at the current state, *Action* represents the action that x affects.

Rule (5.1) says that if χ holds, then *Action* is redundant at current state, thus should not occur at this state. That means only if χ is $false$, *Action* may occur. Otherwise, *Action* should not take place.

ESK provides extra preconditions for the related actions. To encode the knowledge into the original program, we simply put *not* χ into the set of the *Action*'s preconditions. The only rules that are affected by ESK are action rules. $(ESK-1)$ is an encoding for (5.1):

$$\begin{aligned} &HeadAction(Obj,..,T) \leftarrow \\ &\quad preconditions, \qquad (ESK-1) \\ &\quad not\ \chi. \end{aligned}$$

$HeadAction(Obj, .., T)$ is the head of action rule in the original program which could be $Action(Obj, .., T)$ or $\{Action(Obj, .., T)\}$.

ESK will prune search space. After applying ESK, the action rules are changed to contain more preconditions. The extra precondition from ESK is used to prune search space in the following way:

Suppose χ represents the initial or goal property of objects represented by variable Obj and obj_1 is one of the objects. Suppose χ for obj_1 is *true*. Before applying ESK, all the atoms instantiated from $Action(obj_1, .., T)$ are choice points. After applying ESK, atoms instantiated from $Action(obj_1, .., T)$ are $false$. This way, the redundant actions $Action(obj_1, T)$ are pruned. The head atoms of any rule whose body contains action $Action(obj_1, T)$ will never become *true*, thus the search trees with redundant $Action(obj, T)$ as root are pruned at a very early stage of the search.

Since the actions pruned by ESK are redundant, and optimal plans do not contain any redundant actions, the above encoding of ESK will preserve optimal plans for the underlying planning problem.

5.3 State relationship knowledge

During the construction of a plan, the relationships among the properties of two or more states can work as pruning knowledge. As we stated before, there are two subclasses for state relationship knowledge: one is the relationships among the properties of some states; the other consists of the relationships among state conditions and actions. We discuss them one by one, in detail, in the following two subsections.

5.3.1 Relationships about state properties

The relationship among the properties of some states is the effect of a set of actions. Some relationships should not exist, since all the action sets that construct the relationship introduce redundant actions to plans.

Recall the example discussed in Section 5.1: the relationship "return a ball back to its original room", which can be described as a set of properties $\{at(A, R, T)$, *not* $at(A, R, T_1)$,$at(A, R, T_2)\}$ where $T_1 = T + 1$ and $T_2 > T_1$, is such a "bad" relationship. The relationship can be constructed by action set $\{pickup(G, A, T)$, $putdown(G, A, T_1)\}$. Before $pickup(G, A)$ at state T, the property of ball A is $at(A, R)$. After the action *pickup*, the property of A is changed to $on(G, A)$ which makes *not* $at(A, R)$ *true* at state T_1. Then, after action $putdown(G, A, T_1)$ takes place, the property of A changes back to $at(A, R)$ again. The action set is redundant in the current plan, since its effect is nothing, and its removal from a plan can result in another shorter plan.

The same relationship can be constructed by different sets or sequences of actions. For example, the above relationship can also be constructed by the following action sets:

$\{pickup(G, A, T), move(robot, R, R_1, T + 1),$
$move(robot, R_1, R, T + 2), putdown(G, A, T + 3)\}$

$\{pickup(G, A, T), move(robot, R, R_1, T + 1), move(robot, R_1, R, T + 2),$

$move(robot, R, R_1, T+3), move(robot, R_1, R, T+4), putdown(G, A, T+5)\}$

Destroying the relationship will prune all the action sets that construct the relationship. This has the same effect as explicitly preventing all the action sets, but is more compact and easier, as it is sometimes difficult to enumerate all the action sets that can lead to the relationship.

The following rule can be used to formalize the knowledge to prevent the relationships:

$$\chi_1, ..., \chi_n \; CANNOT \; COEXIST \qquad (5.2)$$

Where $\chi_1, ..., \chi_n$ together construct the "bad" relationship, χ_i can be a predicate or the negation of a predicate which describes properties of a state. In terms of the stable model, rule (5.2) means that all $\chi_1, ..., \chi_n$ cannot be *true* at the same stable model.

Rule (5.2) can be encoded as an extra constraint added into the original encoding of the underlying planning domain:

$$\leftarrow \chi_1, ..., \chi_i, ..., \chi_n. \qquad (SRK-2)$$

Any SRK that can be formalized as rule (5.2) can prune search space after it is encoded as the above constraint, and added to the original encoding of the planning problem. As the knowledge is encoded as a constraint, it provides or tightens the dependency among the predicates in the constraint.

Say $\{a_1, .., a_m\}$ is one of the action sets that construct the relationship represented by $\{\chi_1, ..., \chi_n\}$. Before applying SRK, when $a_1, .., a_m$ take place, the relationship is constructed by $\chi_1, ..., \chi_n$ at the next state. Because in the original encoding, there is no constraint as $(SRK-2)$ to block the relationship, the search will continue until it finds the given length is too short. After applying SRK, if all actions in $\{a_1, .., a_m\}$ are assigned value *true*, $\chi_1, ..., \chi_n$ will all be *true* and that violates the constraint $(SRK-2)$. The search will backtrack and the action set $\{a_1, ..., a_m\}$ will be pruned. Therefore SRK rules out action sets that introduce redundant actions at early stage.

5.3.2 Relationships among state conditions and actions

Another subclass of SRK concerns relationships among states and actions. These relationships provide additional conditions under which actions may take place.

The SRK of this subclass has the same effect as ESK described in Section 5.2. It consists of the knowledge providing the information: in some intermediate states, if properties of some states are *true*, certain actions should not take place.

We use the following (5.3) to formalize this class of SRK:

$$IF \; \chi \; THEN \; not \; Action \qquad (5.3)$$

In (5.3), χ is a predicate representing a relationship among conditions of some states. Rule (5.3) means that if the relationship represented by χ does not exist, then *Action* MAY take place. Otherwise, it MUST not happen.

This kind of knowledge tightens the conditions under which an action could occur. In the original encodings for the planning problem, all the preconditions are basic. The SRK will add more conditions or constraints in order to control that only actions which can lead to good plans are added into the final plan.

Rule (5.3) is encoded in the following form

$$\begin{aligned} &HeadAction \leftarrow \\ &\quad preconditions, \\ &\quad not\ \chi. \end{aligned} \qquad (SRK-3)$$

$HeadAction(T)$ is the head of action rule in the original program.

We have the following observations: let program P_1 be

$$h \leftarrow Body,\ not\ x.$$

When x is $false$, the value of h depends on the value of *Body*. When x is *true*, h must be $false$. The following program P_2 represents the same relationship between h and x :

$$\begin{aligned} h &\leftarrow Body. \\ \leftarrow &\ h,\ x. \end{aligned}$$

These two programs have the same set of stable models given the value of *Body*.

According to the above observations, the ESK formalized by (5.1) and the SRK formalized by (5.3) can also be encoded as constraints in the following form:

$$\leftarrow Action,\ \chi. \qquad (SRK-4)$$

5.4 Procedural knowledge

Procedural knowledge extracts sequential information from the sequence of actions in final plans. This type of domain knowledge is powerful because, instead of checking preconditions, it provides a shortcut for deriving new actions directly from an "existing" action. It can also provide extra constraints among actions to guarantee the right sequence in the final plans.

Definition 5.2 Let $Action_i$, $Action_j$ be two actions which occur at state i and state j, respectively, where $j > i$, the state distance of $Action_i$ and $Action_j$ is $i - j$, denoted as $dis(i, j)$.

There are three classes of procedural knowledge.

1. **Procedural knowledge with known state distance**

When specifying the sequence of actions, the state distance between the actions is known. For instance, "action *pickup* should occur TWO steps ahead of action *putdown*", the state distance is two.

Depending on the dependency relationships among the actions in a sequence, the procedural knowledge with known state distance can be further divided into two subclasses. One consists of the procedural knowledge that all the actions are dependent on each other. Once all except one action in the sequence occur, this one action must occur in the same plan. The dependency relationship among the actions is mutual. We call this class of knowledge *total procedural knowledge with known state distance.*

The other subclass consists of the procedural knowledge where some actions can determine the existence of an action, but they do not depend on this action. The dependency relationship is single-directed. We call this class of knowledge *partial procedural knowledge with known state distance.*

2. **Procedural knowledge with unknown state distance**

The procedural knowledge does not provide the state distance between actions. For example, the procedural knowledge "action *pickup* should occur **BEFORE** the action *putdown*" does not provide the information about how many steps ahead.

3. Procedural knowledge with unknown objects

This type of procedural knowledge can provide state distance information, and the actions in the sequence are known, however, some objects that the actions are performed on are not specified. This type of knowledge specifies that a set of actions should take place some states after or before another set of actions. We call this type knowledge *procedural knowledge with unknown objects.*

For example, "for any two different locations Loc_1 and Loc_2, $board(Car, Loc_1, T_1)$ must take place TWO states ahead of action $unboard(Car, Loc_2, T_2)$." The state distance is two, but the two locations cannot be specified by the knowledge. This rule expresses that once action $board(Car, Loc_1, T_1)$ occurs, any action in the set $unboard(Car, Loc_2, T_2)$ where $Loc_2 \neq Loc_1$ must take place two states later.

We discuss in detail the formalization and encoding of these four subclasses in the following four subsections.

5.4.1 Total procedural knowledge with known state distance

Let $action_i(T_i)$ represent $Action_i$ which occurs at state T_i. The notation,

$$c: \ action_i(T_i) \prec_k action_j(T_j)$$

denotes that under the condition represented by c :

1. Once one action (either $action_i$ or $action_j$) occurs in a plan, another action must occur in the same plan;

2. Once these two actions take place in the same plan, the order must be that $action_j$ occur k states later than $action_i$; that is, their state distance must be $dis(T_i, T_j) = k$, where k could be a number or a numerical expression.

Notation $n(action_1(T)|\cdots|action_i(T))m$ denotes that of all the i actions, l actions in the list may occur at state T where $n \leq l \leq m$).

Rule

$$c:\ \chi_1(T_1) \prec_{k_{1,2}} \chi_2(T_2) \prec_{k_{3,2}} \cdots, \prec_{k_{(n-1),n}} \chi_n(T_n) \qquad (5.6)$$

is used to formalize the *total procedural knowledge with known state distance.* In rule (5.6), $\chi_i(T_i)$ can be of the form $action_i(T_i)$ or $n(action_1(T)|\cdots|action_m(T))m$. This rule formalizes that under the condition represented by c, once $n-1$ elements in the list occur in a plan in the specified sequence, the *nth* element must occur in the same plan in order to keep the sequence specified by (5.6), and all the elements must occur in the specified order with specified state distances.

For example, in the grippers problem, the rule

$$at(robot, r, 0):\ pickup(left, a, 0) \prec_1 move(robot, r, r_1, 1)$$

means that when the robot stays in room r at state 0, if the left gripper picks up ball a at state 0, the robot should move from room r to another room r_1 at state 1; If the robot moves from room r to another room r_1 at state 1, then the left gripper should pick up ball a at state 0. Another rule,

$$:\ pickup(G, A, T_1) \prec_2 putdown(G, A, T_2)$$

means that under any condition, a gripper *picking up* a ball is two steps ahead of the same gripper *putting down* the ball.

To encode rule (5.6), for each element $\chi_i(T_i)$, construct the following rule:

$$\begin{array}{l} \chi_i'(T_i) \leftarrow \\ \quad \chi_1'(T_1), ..., \chi_{i-1}'(T_{i-1}), \chi_{i+1}'(T_{i+1}), ..., \chi_n'(T_n), \qquad (PK-6) \\ \quad c, T_2 - T_1 = k_{1,2}, ..., T_{n-1} - T_n = k_{(n-1),n}. \end{array}$$

where if $\chi_i(T_i)$ is of the form $action_i(T_i)$, $\chi_i' = \chi_i$. If χ_i is of the form

$$n(action_{j_1}(T)|\cdots|action_{j_q}(T))m$$

$\chi_i' = n\{action_{j_1}(T), \cdots, action_{j_q}(T)\}m$.

For example: PK rule

$$\begin{array}{l} at(robot, r, T_1): \\ \quad pickup(left, a, T_1) \prec_1 move(robot, r, r_1, T_2) \prec_1 putdown(left, a, T_3) \end{array}$$

is encoded as:

$$\begin{array}{l} pickup(left, a, T_1) \leftarrow \\ \quad move(robot, r, r_1, T_2), \\ \quad putdown(left, a, T_3), \\ \quad at(robot, r, T_1), \\ \quad T_2 - T_1 = 1, T_3 - T_2 = 1. \end{array}$$

$$
\begin{aligned}
&putdown(left, a, T_3) \leftarrow \\
&\quad move(robot, r, r_1, T_2), \\
&\quad pickup(left, a, T_1), \\
&\quad at(robot, r, T_1), \\
&\quad T_2 - T_1 = 1, T_3 - T_2 = 1.
\end{aligned}
$$

$$
\begin{aligned}
&move(robot, r, r_1, T_2) \leftarrow \\
&\quad pickup(left, a, T_1), \\
&\quad putdown(left, a, T_2), \\
&\quad at(robot, r, T_1), \\
&\quad T_2 - T_1 = 1, T_3 - T_2 = 1.
\end{aligned}
$$

Procedural knowledge rule

$$
\begin{aligned}
&(at(robot, r, 0) : (pickup(left, a, 0) \mid pickup(left, b, 0)) \prec_1 \\
&move(robot, r, r_1, 1)
\end{aligned}
$$

can be encoded as:

$$
\begin{aligned}
&1\{pickup(left, a, 0),\ pickup(left, b, 0)\}1 \leftarrow \\
&\quad move(robot, r, r_1, 1), \\
&\quad at(robot, r, 0).
\end{aligned}
$$

$$
\begin{aligned}
&move(robot, r, r_1, 1) \leftarrow \\
&\quad 1\{pickup(left, a, 0),\ pickup(left, b, 0)\}1, \\
&\quad at(robot, r, T).
\end{aligned}
$$

5.4.2 Partial procedural knowledge with known state distance

The following formal rule is used to represent the partial procedural knowledge with known state distance:

$$c : \chi_1(T_1) \preceq_{k_{1,2}} \chi_2(T_2) \preceq_{k_{3,2}}, \ldots, \preceq_{k_{(n-1),n}} \chi_n(T_n) \quad (5.7)$$

where $\chi_j(T_j)$ can be of the form $action_i(T_i)$ or $n(action_1(T)|\cdots|action_l(T))m$. $k_{i,i+1}$ is a number representing the state distance of T_i and T_{i+1}. c represents the condition under which the sequence exists.

The above rule denotes that:

1. If all the actions in the sequence specified by rule (5.7) occur in a plan, their state distances must be the same as specified by rule (5.7);
2. If the action(s) represented by $\chi_1(T_1)$ occur(s) at state T_1 in a plan, action(s) represented by $\chi_2(T_2)$ must occur in the same plan $k_{1,2}$ states later; while if the action(s) represented by $\chi_2(T_2)$ occur(s) at state T_2 in a plan, action(s) represented by $\chi_1(T_1)$ do not have to occur in the same plan $k_{1,2}$ states earlier.
3. If, for all j where $1 \leq j \leq i$, all actions represented by $\chi_j(T_j)$ occur at state T_j in a plan in the sequence specified by the rule, then action(s) $\chi_{i+1}(T_{i+1})$ must occur in the same plan $k_{i,i+1}$ states later.

Partial procedural knowledge provides a single directed dependency relationship between some actions. The action(s) at the right side of $\preceq$ depend on the action(s) at the left side, but the action(s) at the left side do not depend on the action(s) at the right side. This single directed dependency relationship is useful for pruning search space in planning. Once the left side action(s) occurs in a plan, we could directly add the right side action(s) into the same plan. Partial procedural knowledge provides one short cut, while total procedural knowledge provides more.

For example, procedural knowledge in the gripper problem

$$(\ goalRoom(R_2), at(robot, R_1, T_1)) : pickup(G, A, T_1) \preceq_1 \\ move(robot, R_1, R_2, T_2)$$

means that if the robot picks up ball A at state T_1, it must move to the ball's goal room at the next state. However, it is not always true that if the robot moves to the goal room, it must pick up A at the previous state, since this room is the destination of many balls.

The encoding of partial PK rule (5.7) is that for each $\chi_i(T_i)$ where $2 \leq i \leq n$, construct the following rule:

$$\chi'_i(T_i) \leftarrow \\ \chi'_1(T_1), ..., \chi'_{i-1}(T_{i-1}),\ c, \qquad (PK-7) \\ T_2 - T_1 = k_{1,2}, ..., T_i - T_{i-1} = k_{i-1,i}.$$

where $\chi'_i(T_i) = \chi_i(T_i)$ if $\chi_i(T_i)$ is of the form $action_i(T_i)$, while if $\chi_i(T_i)$ is of the form $n(action_1(T)|\cdots|action_l(T))m$, $\chi'_i(T_i) = n\{action_1(T), \cdots, action_l(T)\}m$.

5.4.3 Procedural knowledge with unknown state distance

Let

$$action_1(T_1) \Rightarrow action_2(T_2)$$

denote that once actions $action_1$ and $action_2$ occur in a plan under condition c, then action $action_1$ must occur BEFORE $action_2$, but the state distance of these two actions is unknown.

Let χ_i denote an action $action_i(T_i)$ or a set of action $n(action_1(T_1)|...|action_l(T_l))m$. The following rule formalizes the procedural knowledge with unknown state distance.

$$c\ :\ \chi_1(T_1) \Rightarrow \cdots \Rightarrow \chi_n(T_n) \qquad (5.8)$$

Rule (5.8) means that under condition c, once all elements $\chi_i(T_i)$ take place in a plan, all elements must be in a sequence that state T_i is ahead of state T_j when $j > i$.

For example, the rule

$$:\ pickup(G, A, T_1) \Rightarrow putdown(G, A, T_2)$$

represents the procedural knowledge that gripper G should *pickup* ball A before G *puts* it *down* under any condition, thus $T_1 < T_2$.

Since the state distance is unknown, we cannot use the same encoding for PK rules with known state distance. Even for a simple rule such as

$$: \ pickup(G, A, T_1) \Rightarrow putdown(G, A, T_2).$$

If $pickup(G, A, T_1)$ occurs at state T_1, we only know that some state later than T_2, action $putdown(G, A, T_2)$ will take place, but exactly in which state, the PK rule does not indicate. To encode this type of PK rule, constraint is a good choice. Constraint could express the logic that if action $pickup(G, A, T_1)$ occurs, action $putdown(G, A, T_2)$ cannot occur before T_1, or vice versa.

The encoding of PK rule (5.8) is constructed as the following constraint:

$$\begin{array}{ll} \leftarrow \ \chi_i'(T_1), ..., \chi_n'(T_n), c, & (PK-8) \\ \quad T_1 > T_2, ..., T_i > T_j, T_{n-1} > T_n. & \end{array}$$

where $\chi_i'(T_i) = \chi_i(T_i)$ if it is of the form $action_i(T_i)$, if $\chi_i(T_i)$ is of the form $n(action_1(T)|\cdots|action_l(T))m$, $\chi_i'(T_i) = n\{action_1(T), \cdots, action_l(T)\}m$.

5.4.4 Procedural knowledge with unknown objects

The notation

$$c: \ action_i(T_i) \mapsto_k action_j(T_j)$$

denotes that $action_i(T_i)$ must occur k states ahead of $action_j(T_j)$, but there are unspecified objects in these two action predicates. c represents the conditions under which this sequence applies.

The procedural knowledge with unknown objects is formalized by the following rule

$$c: \ \chi_1(T_1) \mapsto_{k_{1,2}} \chi_2(T_2) \mapsto_{k_{2,3}} \cdots \mapsto_{k_{n-1,n}} \chi_n(T_n) \qquad (5.9)$$

In (5.9), $\chi_i(T_i)$ can be of the form $action_i(T_i)$ or $n(action_1(T)|\cdots|action_l(T))m$.

To encode this type of knowledge, we use constraint, since there are unknown objects in the predicates. The constraint is constructed as follows:

$$\begin{array}{ll} \leftarrow \ \chi_1'(T_1), \ \chi_2'(T_2), ..., \ \chi_n', & \\ \quad c, \ T_2 - T_1 \ \neq k_{1,2}, ..., & (PK-9) \\ \quad T_n - T_{n-1} \ \neq k_{n-1,n}. & \end{array}$$

where if $\chi_i(T_i)$ is of the form $action_i(T_i)$, $\chi_i'(T_i) = \chi_i(T_i)$. While, if $\chi_i(T_i)$ is of the form $n(action_1(T)|\cdots|action_l(T))m$, $\chi_i'(T_i) = n\{\chi_1(T), \cdots, \chi_l(T)\}m$.

The procedural knowledge can preserve optimal plans, since the sequence specified is the sequence of actions in the optimal plans. Therefore all the plans the procedural knowledge prunes are not optimal plans.

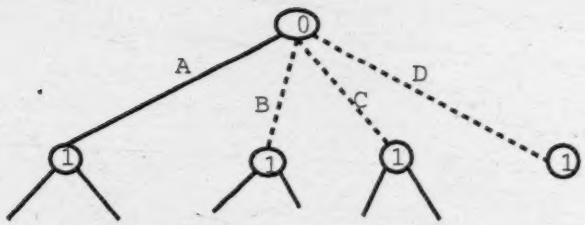


Figure 6: An example to show symmetry breaking can prune search space

5.5 Symmetry knowledge

We do not need to search all isomorphic plans since they can be retrieved using symmetries (by exchanging symmetric objects). Therefore of all isomorphic plans, only one of them should be found by a search procedure. If the search procedure has proved or found that a partial assignment will not lead to a valid plan, to search other isomorphic assignments is a waste of time, since none of the symmetric assignments will lead to a valid plan. Those symmetries will then give rise to an (often exponential) amplification of the search space. To break symmetries is therefore to ensure that whenever a partial assignment is shown not to lead any plan, no symmetric assignment is ever tried. If all the plans are needed, only distinct ones are searched, never symmetric ones. Hence, symmetry-breaking promises to efficiently prune the search space.

An *action tree* is a tree with a state as node and a possible action as arc. The child of a node is the next state after taking the action on the arc. If we could find a leaf that represents the goal conditions, then from the root to the leaf, the actions on the arcs make up a valid plan.

Figure 6 gives a small example which uses action tree to explain how symmetry prunes search space. In the figure, ball a, b, c, d are symmetric objects, action *pickup* could work on any one of them, and the subtrees are isomorphic. If "the left gripper pickup ball a at state 0" will lead to a valid plan M_1, "the left gripper pickup ball b (or c or d) at state 0" will lead to an isomorphic plan M_2, which could be obtained by exchanging ball b (or c or d) with ball a from plan M_1. If "the left gripper pickup ball a at state 0 " will not lead to a valid plan, "the left gripper pickup ball b (or c or d) at state 0" will also not lead to a valid plan. Therefore, if we detect that ball a, b, c, d are symmetric objects, we should search only one subtree, thus saving 75% of the search space.

Many problems have symmetries due to indistinguishable symmetric objects. Transforming symmetric objects to non-symmetric objects will break the symmetries generated by the objects. In order to avoid searching symmetric space, we must solve two problems: discovering symmetric objects and breaking them.

5.5.1 Symmetry discovery

In planning, symmetric objects are not always symmetric. When the conditions change, the effects of actions on these objects will change them from being symmetric to non-symmetric, or vice versa. Therefore object symmetry is state dependent.

Symmetric objects are also action-dependent. Two objects are symmetric because when an action works on either of them first, the final plan is isomorphic to the plan where the action works on the other first. This is why when we break the symmetries, the action rules will be involved.

Let obj_1, obj_2 be two objects in a planning problem. They are symmetric at the current state T if they have the following properties:

1. If their properties are considered at the goal state, their properties that must hold at the goal state must be the same, and must not hold at the current state;
2. They are from the same domain (that is, of the same type);
3. Their properties which hold at the current state T are the same;
   For example, if they are balls in the gripper problem, they are in the same room, not being held by the robot. If they are trucks at the logistics problem, they are located at the same location within the same city, and there is no package loaded on them.
4. A same action cannot perform on obj_1 and obj_2 at the same state, but the different orders (of the action working on them) will derive isomorphic plans. This is essential, since if $obj_1$and obj_2 can be "performed" by the same action at the same state, we do not need to differentiate them.

For example, in the gripper problem, there are two classes of objects: grippers (left and right) and balls. Both of these two types of objects are possible symmetric objects. The balls A and B could be symmetric objects when :

1. The properties which are required to hold at the goal state are the same: staying in the goal room. At the current state, they are both in the initial room;
2. They are of the same type (both are balls);
3. Their properties at the current state are the same: they are located in the same room- the initial room, not being held by any grippers;
4. Action $pickup(Gripper, Ball, T)$ cannot perform on two balls at the same time since one gripper can hold only one ball at a time. However the plan where the gripper first picks up ball A is isomorphic to the plan where the same gripper first picks up ball B.

To encode the detection of symmetry, a new predicate $symmetric(objType, obj_1, obj_2, T)$ is constructed for each type of symmetric object. The encoding is straightforward as $(SK-8)$: the head of the rule is $symmetric(objType, obj_1, obj_2, T)$, the body consists of property 1-3 for obj_1 and obj_2. Property number 4 cannot be encoded using predicate or inference rules. It depends on the judgment of users.

$$
\begin{aligned}
&symmetric(objType, A, B, T) \leftarrow \\
&\quad objType(A), \\
&\quad objType(B), \\
&\quad state(T), \\
&\quad goal\_properties(A), \qquad (SK-8) \\
&\quad goal\_properties(B), \\
&\quad current\_properties(A, T), \\
&\quad current\_properties(B, T).
\end{aligned}
$$

The properties of A and B which hold at the current state T must not be same as their properties which are required to hold at the goal state.

Take object *ball* in the gripper problem as an example, we construct the following rule to detect symmetric objects:

$$
\begin{aligned}
&symmetric(ball, A, B, T) \leftarrow \\
&\quad ball(A), ball(B), state(T), \\
&\quad initRoom(R_1), \\
&\quad R_1 \neq R_2, \\
&\quad goalAt(A, R_2), \\
&\quad goalAt(B, R_2), \\
&\quad at(A, R_1, T), \\
&\quad at(B, R_1, T).
\end{aligned}
$$

where $ball(A)$ and $ball(B)$ correspond to property number 2. Since there are only two rooms, R_2 which is not the initial room must be the goal room. $initRoom(R)$, $at(A, R, T)$ and $at(B, R, T)$ are properties that hold at the current state T for A and B. $goalAt(A, R_2)$ and $goalAt(B, R_2)$ are properties that must hold at the goal state. Since ball A and B are in the initial room, their properties that must hold at the goal state do not hold at the current state.

5.5.2 Symmetry breaking

There are several ways to break symmetry. Symmetry breaking during search (SBDS) [17] is a recent development which adds constraints during search, to ensure that any assignment symmetric to one already considered will not explored. SBDS is not appropriate here since we do not change or modify the planning algorithm.

Another way to break symmetry is to add constraints to a program in order to require it to have only one of the symmetric solutions [34]. Adding extra constraints is applicable for our purpose, so we will explore this method in detail.

In CSP, if variables are detected to be symmetric, e.g., x_1, x_2 and x_3, these variables are formed into a list and ordered lexicographically [13]. Symmetry is broken by adding weak inequalities between adjacent variables in the list, e.g., $x_1 \leq x_2$, $x_2 \leq x_3$. When taking a variable from the list, the CSP will choose the smallest or the largest one, according to these added constraints.

The same method can be used to break symmetry for planning problems encoded in logic programming. After detecting that some objects are symmetric, they are first

sorted, and each of them is assigned an order. During the search, when taking an object to perform an action, only the smallest, or one of the smallest, is chosen.

Since the instance information does not provide information about the order of the objects, new predicates are constructed to assign each symmetric object an order. While assigning order to symmetric objects at intermediate states is complicated, we choose to order all objects if the type of object has the possibility to be symmetric. For example, we know in some states, object *ball* in the gripper problem becomes symmetric, so we list all balls in ascending order.

Let

$$objType: a_1, a_2, \ldots, a_n$$

represent a list with elements $a_1, a_2, \ldots, a_n$ from domain $objType$. A list can be represented using the following predicates in logic programming:

$$head(objType, a_1).$$
$$list(objType, a_2, a_1).$$
$$\ldots$$
$$list(objType, a_n, a_{n-1}).$$

where $head(objType, a_1)$ represents that the list contains the objects of type $objType$ and the head of the list is a_1, $list(objType, a_2, a_1)$ tells that in the list a_2 follows a_1.

Using this expression, every object in the list can be given a number as its order. The following constructs this process:

$$order(objType, A, 1) \leftarrow head(objType, A). \qquad (SK-9)$$

$$order(objType, B, N+1) \leftarrow list(objType, B, A), order(objType, A, N). \qquad (SK-10)$$

In the above rules, rule $(SK-9)$ is used to initialize the order of the head element in the list as one. $(SK-10)$ gives an incremental number to the element in the list, and the last element gets the largest number. These rules guarantee that each object in the list gets a distinct number.

To get the object with the smallest number in the list, the following rules are constructed:

$$greater(objType, A, B, T) \leftarrow symmetric(objType, A, B, T), order(objType, A, N_a), order(objType, B, N_b), N_b > N_a. \qquad (SK-11)$$

$$nonLess(objType, 1, A, T) \leftarrow symmetric(objType, A, B, T), \qquad (SK-12)$$

$greater(A, B, T)$,
$A\ !=\ B$.

$least(objType, 1, A, T) \leftarrow$
$not\ nonLess(objType, 1, A, T)$, $\quad (SK-13)$
$symmetric(objType, A, B, T)$.

In the above rules, the predicate $greater(objType, A, B, T)$ in rule $(SK-11)$ is used to compare two symmetric objects, A and B, in terms of their order. Rule $(SK-12)$ checks if there is an object whose order is greater than object A. If this is not the case, object A is the least, as expressed by rule $(SK-13)$.

When symmetries are to be broken, the action rule for the action that is performed on the symmetric object will be changed. The smallest one of all the symmetric objects should be chosen, not any one. An extra condition will be added to the body of the action rule as

$HeadAction(Obj, SymObj, ..., T) \leftarrow$
$preconditions$, $\quad (SK-14)$
$least(objType, 1, SymObj, T)$.

where *SymObject* represents a symmetric object, and *Body* represents the original body literals of the original action rule. $HeadAction(Object, SymObject, ..., T)$ represents the head of the original action rule. The action encoded by $(SK-14)$ will only be performed on the smallest symmetric object, while excludes all other symmetric objects.

It is a straightforward assumption that by breaking more symmetries, we can further reduce the search space. However, as the number of symmetries detected increases, so does the overhead. Sometimes, the extra overhead can outweigh the benefit of reducing search space. Therefore, breaking all the symmetries in the problem is not always an optimal strategy; sometimes, we leave some symmetries in the program.

If there are symmetries left, or parallel actions in the same state can be performed on the same set of symmetric objects, more than one symmetric object should be allowed to choose at one state. In this case, we need to construct rules to identify the least N objects $(N > 1)$.

For example, in the gripper problem, if we decide not to break the symmetries created from two symmetric grippers, the $pickup(Gripper, Ball, T)$ action need two balls to choose from, since there are two *Grippers* which can perform the *pickup* actions at the same state. The action rule encoded as rule $(SK-14)$

$\{pickup(G, A, T)\} \leftarrow$
$free(G, T)$,
$at(robot, R_1, T)$,
$at(A, R_1, T)$,
$least(A, T)$,
$not\ goal(T)$.

only allows the smallest *ball* to be picked. This makes the *pickup* action non-parallel.

The rules to get N smallest symmetric objects are $(SK-11)$, $(SK-12)$, $(SK-13)$ plus the following new rules:

$$
\begin{aligned}
&nonLess(objType, N, A, T) \leftarrow \\
&\quad symmetric(A, C, T), \\
&\quad not\ least(objType, N-1, A, T), \\
&\quad not\ least(objType, N-1, C, T), \\
&\quad greater(A, C, T), \\
&\quad C\ !=A.
\end{aligned}
\qquad (SK-15)
$$

$$
\begin{aligned}
&least(objType, N, A, T) \leftarrow \\
&\quad not\ nonLess(objType, N, A, T). \\
&\quad not\ least(objType, N-1, A, T).
\end{aligned}
\qquad (SK-16)
$$

$$
\begin{aligned}
&least(objType, N, A, T) \leftarrow \\
&\quad least(objType, N-1, A, T).
\end{aligned}
\qquad (SK-17)
$$

Let $\{a_1, \ldots, a_n, a_{n+1}, \ldots, a_m\}$ be the list of the symmetric object in an ascending order. The predicate $least(objType, N, A, T)$ represents that symmetric object A is in the list $\{a_1, \ldots, a_n\}$. The predicate $nonLess(objType, N, A, T)$ represents that object A is not in the list $\{a_1, \ldots, a_{n-1}\}$ and it is not the smallest object in $\{a_n, \ldots, a_m\}$, i.e., it is not a_n. Rule $(SK-15)$ uses these conditions to check if object A is in $\{a_n, \ldots, a_m\}$ but not a_n:

1. $symmetric(objType, A, C, T)$: Object A and C are symmetric object at state T
2. $not\ least(objType, N-1, A, T)$: A is not in the list $\{a_1, \ldots, a_{n-1}\}$, then A is in $\{a_n, \ldots, a_m\}$
3. $not\ least(objType, N-1, C, T)$: C is not in the list $\{a_1, \ldots, a_{n-1}\}$, then C is in $\{a_n, \ldots, a_m\}$
4. $greater(A, C, T)$: the order of Object A is greater than that of object C, then A is not a_n

If all these conditions are satisfied, we know that A is in the list $\{a_n, \ldots, a_m\}$ but not a_n, then $not\ nonLess(objType, A, N, T)$ must be a_n, if A is not in the list $\{a_1, \ldots, a_{n-1}\}$. That is precisely what rule $(SK-16)$ expresses.

Rule $(SK-17)$ represents that if $A \in \{a_1, \ldots, a_{n-1}\}$, then $A \in \{a_1, \ldots, a_{n-1}, a_n\}$.

5.6 Distance knowledge

If the lower bound needed to change properties of an object (or all objects of a type) which hold at the current state to properties which hold at the goal state is less than the available steps, then the current partial plan cannot lead to a plan within the given number of steps. For example, if the given length of plan is 10 and the current

state is state 2, then the number of available steps is 8. If the lower bound to move a package from a location L to its goal location L_1 is 9, then $9 > 8$ and the planner should stop the current search branch immediately.

To compute the lower bound, we construct a rule such as the followings:

$$
\begin{array}{ll}
lowerBound(Obj, N, T) \leftarrow & \\
\quad current\_conditions(Obj, T), & (DK-19) \\
\quad goal\_conditions(Obj), & \\
\quad facts, & \\
\quad N = formaluToComputeLowerBound. &
\end{array}
$$

where $formulaToComputeLowerBound$ is a formula to compute the lower bound.

To stop the search once the lower bound is greater than the steps available, we construct the following rules:

$$
\begin{array}{ll}
f \leftarrow & \\
\quad lowerBound(Obj, N, T) > (steps - T). & (DK-20) \\
\leftarrow f. & (DK-21)
\end{array}
$$

If the lower bound is greater than the remaining steps, which is computed by $steps - T$, then atom f is $true$, which will produce a conflict with $(DK-21)$ which says f must be $false$. Then the search should stop extending the current branch and backtrack.

5.7 Related work on domain dependent knowledge

Procedural knowledge has been used in GOLOG, a logic programming language for agent programming, control, and execution, based on a situation calculus theory of actions [22]. GOLOG has been primarily used as a programming language for high-level agent control in dynamic environments. More recently, GOLOG has been used for general planning [12].

When it is used for planning, a GOLOG program specifies an arbitrarily incomplete plan that includes non-deterministic choice points. For example, $p_1; p_2; (u_1|u_2|u_3); f?$ represents plans which have action p_1 followed by action p_2, then followed by one of actions u_1, u_2, or u_3, in order to make fluent f true in the plan. GOLOG then merely needs to decide which one of u_1 ,u_2, or u_3 is chosen to make a valid plan.

The range of procedural knowledge described in this thesis is far wider than that used in GOLOG. The knowledge in this thesis does not only provide sequential information for specific actions, it can also specify the sequence of a set of actions, even the sequence of the whole plan can be specified.

Distance knowledge is first used by CPlan [4]. In CPlan language, lower and upper bounds on how many steps needed for a variable to change from one value to another are computed when applying distance knowledge. In CPlan, lower bound and upper bound are computed only at the initial state.

Since, in our encodings, the length is given, the upper bound is out of context. The lower bound is extended to any state except the goal state.

Our approach to making use of domain dependent knowledge differs from previous work. First, domain dependent knowledge will not be encoded into the search algorithm. Unlike CPlan, the information we use does not refer to the underlying planning algorithm; rather it only adds extra information to describe the planning domain. It is up to the planning algorithm to use these extra constraints to control the search. This means that, although the control information we utilize is domain dependent, the user need not know anything about the planning algorithm.

Second, in contrast with TLPlan [1], there is no need for a new language to encode domain dependent knowledge. The domain dependent knowledge is encoded as extra constraints into the original program which encodes the domain independent knowledge of the planning problem. These extra constraints are constructed in the same syntax as in the original programs.

6 Experiments

We have implemented a range of test domains to determine:

1. how simple it is to specify domain dependent knowledge using our encodings, and
2. how effective the domain dependent knowledge is for pruning search space in planning.

In our empirical tests we run Smodels2.26 on a Pentium IV 1.5GHz machine with 1GB RAM. This amount of memory is sufficient for Smodels in all of the tests.

We tested our encodings with five planning domains. The logistics domains [5, 38], the blocks world domains [18] and the gripper domains are from the CPlan testbed. The elevator planning domains are from the AIPS planning competition in 2000 [2] while the ferry domains are from TLPlan distribution [1].

In Section 6.1, we describe the five planning problems briefly. From Section 6.2 to Section 6.6, we show the experimental results for each class of the domain dependent knowledge. At the end of each section, we provide a summary to analyze the experiments for one class of the knowledge, along with some insights. Section 6.7 summarizes all the experiments.

6.1 Five planning problems

All the encodings of the five planning problems are listed in the Appendix. Here we provide brief descriptions in terms of the predicates that are used to describe properties of objects, the preconditions and effects of the actions applied in the planning domains.

Gripper Problem

The gripper problem has been described in Chapter 3.

Blocks World

The blocks world planning problem has been widely investigated by planning researchers, primarily because it appears to capture several of the relevant difficulties posed to planning systems. This description of the blocks world problem is based on [18].

> The objects in the problem domain include a finite number of cubical blocks, and a table large enough to hold all of them. Each block is on a single other object (either another block or the table). For each block b, either b is clear or else there is a unique block a sitting on b. There is one kind of action: move a single clear block, either from another block onto the table, or from an object onto another clear block. As a result of moving b from c onto d, b is sitting on d instead of c, c is clear(unless it is the table), and d is not clear(unless it is the table).

Predicate $on(X, Y, T)$ reads that block X is on object Y at state T. This is the only predicate to express the conditions of the blocks. The only action is represented by predicate $move(X, Y, T)$ which reads "block X moves on top of object Y at state T".

The encoding of the blocks world problem is described in Appendix B.

Logistics World

A popular domain is the logistics world. In this domain, we have two types of vehicles: trucks and airplanes. Trucks can be used to transport packages within a city, and airplanes can be used to transport packages between airports. The problem in this domain usually starts with a collection of packages at various locations in various cities, and the goal is to redistribute these packages to their destinations.

The encoding of the Logistics World problem is in Appendix C. The predicates that are used to describe properties of objects are listed in the following table.

predicat e	meaning
$in(P, Tr, T)$	package P is in *truck* Tr at state T
$at(P, L, T)$	package P is at location L at state T
$in(P, Pl, T)$	package P is in *plane* Pl at state T
$at_t(Tr, L, T)$	truck Tr is at location L at state T
$at_p(Pl, L, T)$	plane Pl is at location L at state T

There are six actions in the problem whose preconditions and effects are listed in the following table.

Action	Preconditions	Effects
$loadTruck(Tr, P, T)$	$at_t(Tr, L, T), at(P, L, T)$, *Tr is not moving*	$in(P, Tr, T+1)$
$unloadTruck(Tr, P, T)$	$in(P, Tr, T), at_t(Tr, L, T)$ *Tr is not moving*	$at(P, L, T+1)$
$driveTruck(Tr, L_1, L_2, T)$	$at_t(Tr, L_1, T)$, $sameCity(L_1, L_2)$	$at_t(Tr, L_2, T+1)$
$loadPlane(Pl, P, T)$	$at_p(Pl, L, T), at(P, L, T)$, *Pl is not flying*	$in(P, Pl, T+1)$
$unloadPlane(Pl, P, T)$	$in(P, Pl, T), at_p(Pl, L, T)$ *Pl is not flying*	$at(P, L, T+1)$
$flyPlane(Pl, L_1, L_2, T)$	$at_p(Pl, L_1, T)$, $airport(L_1)$, $airport(L_2)$	$at_p(Pl, L_2, T+1)$

Ferry Problem

The Ferry World is also a popular planning domain, with actions that board a car onto a single ferry, or unboard a car from it. The ferry can only transport one car at a time, and a *move* action can be applied to move the ferry between any two locations. The goal of the problem is to transport all the cars to their destinations.

This problem is similar to the gripper problem with the difference that, in the gripper problem, all the balls are initially in the same room, and their goal rooms are another room. The robot can therefore move between these two rooms. However in the ferry problem, all the cars have their own initial locations which may be the same, or may be different, and their goal locations may also be different. The ferry will have to move among all these different locations.

The encoding of the ferry problem is presented in Appendix D. The predicates that are used to describe properties of objects are listed in the following table.

predicate	meaning
$at(Car, L, T)$	car Car is at location L at state T
$in(ferry, Car, T)$	car Car is in the ferry at state T
$empty(ferry, T)$	the ferry is empty

There are three actions in the problem whose preconditions and effects are listed in the following table.

Action	Preconditions	Effects
$board(Car, L, T)$	$at(Car, L, T), at(ferry, L, T)$, $empty(ferry, T), not\ goal(T)$, $not\ moving(ferry, T)$	$in(ferry, Car, T+1)$
$unboard(Car, L, T)$	$in(ferry, Car, T)$, $at(ferry, L, T), not\ goal(T)$, $not\ moving(ferry, T)$	$at(Car, L, T+1)$
$move(ferry, L_1, L_2, T)$	$at(ferry, L_1, T)$, $not\ goal(T)$	$at(ferry, L_2, T+1)$

Elevator Problem

In the elevator planning domain, there is an elevator which will transport passengers from their initial floors to their goal floors. There are three objects: the elevator called $lift$, a number of passengers, and several floors. The elevator can move up and down to any floor. When it stops at a floor, any passenger in the elevator can be unboarded, and any passenger waiting at the floor can be boarded on to the elevator. The goal of the problem is to transport all the passengers to their destination floors.

The encoding of the elevator problem is presented in Appendix E. The predicates that are used to describe properties of objects are listed in the following table.

predicate	meaning
$at(P, F, T)$	passenger P is at floor F at state T
$in(lift, P, T)$	passenger P is in the lift at state T
$at(lift, F, T)$	the lift is at floor F

There are three actions in the problem whose preconditions and effects are listed in the following table.

Action	Preconditions	Effects
$board(P,F,T)$	$at(P,F,T), at(lift,F,T)$, $not\ affected(lift,T)$, $not\ goal(T)$	$not\ goal(T)$
$unboard(P,F,T)$	$in(lift,P,T), at(lift,F,T)$, $not\ affected(lift,T)$, $not\ goal(T)$	$at(P,F,T+1)$
$up(lift,F_1,F_2,T)$	$above(F_2,F_1)$, $at(lift,F_1,T)$, $not\ goal(T)$	$at(lift,F_2,T+1)$
$down(lift,F_1,F_2,T)$	$above(F_1,F_2)$, $at(lift,F_1,T)$, $not\ goal(T)$	$at(lift,F_2,T+1)$
$stop(lift,F,T)$	$at(lift,F,T)$, $not\ goal(T)$	$stop\_at(lift,F,T+1)$

In the following sections, the table column labeled "steps" represents the given length of plans, the table column labeled "time(Seconds)" represents the search time to find the first optimal plan. The column with the subtitle "without" contains search time of the logic program encoded for planning domains without any domain dependent knowledge. Another column under title "time(Seconds)" represents the search time of the logic program with the domain dependent knowledge represented by the rule number in it. For example, the column with subtitle "$ESK\ I$" represents the search time of the logic program with end state knowledge represented by $ESK\ I$.

6.2 Experiments in applying end state knowledge

In this section, we apply end state knowledge to several planning domains and compare the search performance before and after applying the knowledge.

Applying ESK to the gripper problem

For the gripper problem, we extract the following two ESK rules. $goalAt(Ball,Room)$ represents that $Room$ is the goal room of $Ball$.

$ESK01$ $IF\ (goalAt(Ball,Room), at(Ball,Room,T))$
$THEN\ not\ pickup(Gripper,Ball,T)$

$ESK02$ $IF\ (not\ goalAt(Ball,Room), at(robot,Room,T))$
$THEN\ not\ putdown(Gripper,Ball,T)$

After applying the above ESK, the goal property of $Ball$ is added to the preconditions of actions *pickup* and *putdown*, and action rules for these two actions are changed as follows:

$\{pickup(Gripper,Ball,T)\} \leftarrow$
$free(Gripper,T)$,
$at(robot,Room,T)$,
$at(Ball,Room,T)$,

#ball	steps	Time(Seconds)	
		without	$ESK01 + ESK02$
2	3	0.02	*0.02*
4	7	0.780	*0.13*
5	11	136.07	*11.02*
6	11	2611.740	*11.89*
7	15	more than 2 days	*2947.93*
8	15	more than 2 days	*6336.35*

Table 2: Experimental results of the gripper problem with/without ESK

instance	#package	steps	Time(Seconds)	
			without	$ESK03 + ESK04$
p04	7	9	2.11	1.31
p01	6	9	1.88	1.29
p02	5	7	3.16	2.39
p05	4	11	70.28	56.07
p07	10	9	more than 2 days	43.11

Table 3: Experimental results of the logistics problem with/without ESK

$$
\begin{aligned}
&\textit{not affected}(\textit{robot}, T),\\
&\textit{not goal}(T),\\
&\textit{not } \texttt{goalAt}(\textit{Ball}, \textit{Room}).
\end{aligned}
$$

$$
\begin{aligned}
\{\textit{putdown}(\textit{Gripper}, \textit{Ball}, T)\} \leftarrow\\
&\textit{at}(\textit{robot}, \textit{Room}, T),\\
&\textit{on}(\textit{Gripper}, \textit{Ball}, T),\\
&\textit{not affected}(\textit{robot}, T),\\
&\textit{not goal}(T),\\
&\texttt{goalAt}(\textit{Ball}, \textit{Room}).
\end{aligned}
$$

Table 2 shows the experiment results before and after applying ESK rules, $ESK01$ and $ESK02$, in the gripper domain.

Applying ESK to the logistics problem

In the logistics domain, we extract two ESK rules formalized as the following rules $ESK03$ and $ESK04$. Rule $ESK03$ represents "*do not load truck if the package is at its goal location.*" Rule $ESK04$ represents "*do not load a package onto an airplane if the package is at its goal city.*"

$ESK03$ $\quad IF\ (goalAt(Package, Loc), at(Package, Loc, T))$
$\quad THEN\ not\ loadTruck(Package, Tr, T)$

$ESK04$ $\quad IF\ (goalCity(Package, City), at(Package, Loc, T), inCity(loc, City))$
$\quad THEN\ not\ loadPlane(Package, P, T)$

#passenger	#floor	steps	Time(Seconds)	
			without	$ESK05 + ESK06$
1	3	3	0.00	0.00
2	4	6	0.00	0.00
3	5	10	1.17	0.13
4	7	12	17.28	2.50
5	9	14	853.86	13.40
6	11	14	3676.65	112.91

Table 4: Experimental results of the elevator problem with/without ESK (have solution)

#passenger	#floor	steps	Time(Seconds)	
			without	$ESK05 + ESK06$
1	2	2	0.00	0.00
2	4	5	0.00	0.00
3	5	9	0.51	0.14
4	7	11	21.40	2.58
5	9	13	932.24	38.71
6	11	13	1563.58	99.22

Table 5: Experimental results of the elevator problem with/without ESK (no solution)

Table 3 shows the experimental results before and after applying the above ESK rules in the logistics domains.

Applying ESK to the elevator problem

The knowledge represented by rules $ESK05$ and $ESK06$ is extracted from the elevator domain. Rule $ESK05$ says "*do not board a passenger if the floor is the passenger's destination.*" Rule $ESK06$ represents "*do not unboard a passenger if the floor is not the passenger's destination.*"

$ESK05$ $\quad IF\ goalAt(Person, Floor)\ THEN\ not\ board(Person, Floor, T)$

$ESK06$ $\quad IF\ not\ goalAt(Person, Floor)\ THEN\ not\ unboard(Person, Floor, T)$

Table 4 shows the experimental results with and without ESK rules in the elevator domains when the given plan length guarantees that the plan is optimal. Table 5 shows the experimental results with and without ESK rules when the given plan length(steps) is too short to get a valid plan.

Applying ESK to the ferry problem

We extract two ESK rules, $ESK07$ and $ESK08$, from the ferry domains. Rule $ESK07$ says "*do not board a car if the location is the car's destination.*" Rule $ESK08$ represents "*do not unboard a car if the location is not the car's destination.*"

$ESK07$ $\quad IF\ goalAt(Car, Loc)\ THEN\ not\ board(Car, Loc, T)$

instance	#car	#location	steps	Time(Seconds)	
				without	$ESK07 + ESK08$
p03	4	4	11	1.04	0.15
p05ss	5	4	12	3.09	0.56
p04	5	4	15	124.44	3.08
p05	6	6	15	149.38	23.07
p04s	4	4	16	150.61	112.50

Table 6: Experimental results of the ferry problem with/without ESK (have solution)

instance	#car	#location	steps	Time(Seconds)	
				without	$ESK07 + ESK08$
p03	4	4	10	0.48	0.10
p05ss	5	4	11	2.72	0.36
p04	5	4	14	45.86	4.86
p05	6	6	14	341.14	24.65
p04s	4	4	15	122.94	36.72

Table 7: Experimental results of the ferry problem with/without ESK (no solution)

$ESK08$ $\quad IF\ not\ goalAt(Car, Loc)\ \ THEN\ not\ unboard(Car, Loc, T)$

Table 6 shows the experimental results with and without ESK rules in the ferry domains when the given plan length(steps) guarantees that the plan is optimal. Table 7 shows the experimental results with and without ESK rules when the given plan length(steps) is too short to get a valid plan.

Applying ESK to the blocks world problem

We extract ESK rule $ESK09$ from the blocks world domains. Rule $ESK09$ says "*do not move a block currently staying on the table if at goal state, it is on table.*"

$ESK09$ $\quad IF\ \ sameAsGoalOnTable(Block, table)\ \ THEN\ not\ move(Block, Y, T)$

Table 8 shows the experimental results with and without ESK rules in the blocks world domains when the given plan length(steps) guarantees that the plan is optimal.

Summary of ESK

We can extract ESK from all the transportation problems. Once an object gets to its goal location, it should not be moved after that. Also, its goal location is not related to the goal locations of other objects. ESK for the blocks world problem is not strong, since the positions of the blocks are interrelated. A block reaches its own goal position does not mean it will stay at that position. It depends on whether all the blocks under it are at their goal positions.

We can summarize the above experimental results as follows:

1. End State knowledge is easy to formalize and encode.

#blocks	plan length	time(Second)	
		without	ESK09
12	11	2.94	2.08
13	12	5.53	4.15
14	14	16.98	14.24
19	10	17.50	12.70
20	10	22.76	10.49
16	8	3.20	2.21
17	9	2424.42	2202.83
18	10	3146.81	2728.86

Table 8: Experimental results of applying $ESK09$ to the blocks world problem

2. End state Knowledge is impressive in its improvement of search efficiency. It improves the search performance by up to two orders of magnitude.

3. The more difficult an instance is, the better ESK works.
For example, in Table 2, when there are 5 balls, the encoding with ESK is 12 times faster than the one without it. When the number of balls is 6, the encoding with ESK is 200 times faster.

6.3 Experiments in applying state relationship knowledge

In this section, we apply state relationship knowledge to several planning domains, and compare the search performance before and after applying the knowledge.

Applying SRK to the blocks world problem

From the Blocks World domain, we extract SRK as "*don't move a block from table to table.*" The knowledge can be formalized as the following rule:

$SRK01$ $\quad IF\ on(X, table, T)\ THEN\ not\ move(X, table, T)$

This SRK rule is encoded as the following constraint:

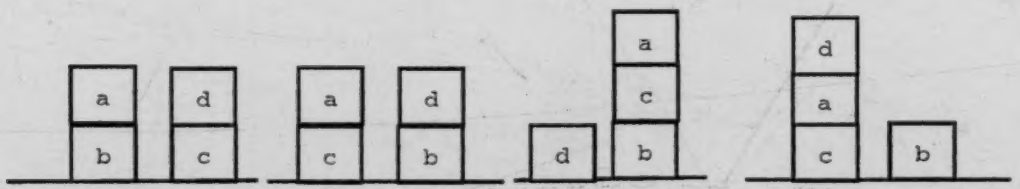
$$\leftarrow\ move(X, table, T), on(X, table, T).$$

Table 11 shows the experimental results before and after applying the above SRK rule in the blocks world domain. The SRK rule can help speed up the search. For small instances, the speedup is not impressive. However, for the hard instances, it can achieve an optimal plan in just one minute, such as the cases for *bw*.17, *bw*.18, and *bw*.20. The speedup is up to 3 orders of magnitude. For the very hard instances, *bw*.21 and *bw*.19, the encodings without SRK had not get a solution after two days, while after applying the SRK rule, one was achieved in 3 minutes.

We extract another SRK rule as "*move a block on top of a good tower*" from the Blocks World domain. A *tower* is a set of blocks sitting on top of one another. In Figure 7, there are two towers in the initial state: $\{a, b\}$ and $\{c, d\}$. A *good tower* is a tower in which all the blocks are at their goal positions.

Instance	#blocks	steps	time(Second)	
			without	SRK01
bw.16	16	8	3.2	3.08
bw.17	17	9	2424.42	7.78
bw.18	18	10	3146.81	53.79
bw.19	19	10	more than 2 days	163.47
bw.20	20	10	22.760	13.140
bw.21	21	11	more that 2 days	24.560

Table 9: Experimental results of applying SRK01 to the blocks world problem




```
Initial State:
on(a,b,0).
on(b,table,0).
on(c,table,0).

Goal State:
goal_on(b,c).
goal_on(a,table).
goal_on(c,table).

non-goal State T1:
on(a,c,T).
on(c,b,T).
on(b,table,T).
on(d,table,T).

non-goal State T2:
on(d,a,T).
on(a,c,T).
on(c,table,T).
on(b,table,T).
```

Figure 7: An example of goal position in the blocks world problem

This is a good strategy to search for good plans since, otherwise, if a block (say a) is moved on top of a "bad" tower, then later, the tower has to be dismantled, so that the block which is not at its goal position can be moved to its goal position; then block a has to be moved back. If we wait until the tower becomes a good tower, then move a block onto it if needed, the tower does not need to be dismantled. The "dismantling" will introduce redundant actions during search, therefore it needs to be prevented.

To express the information that block X is in a good tower, we construct a new predicate $goodTower(X,T)$ which states that block X is in a good tower at state T. The following rules represent the logic about a good tower:

$$
\begin{aligned}
&goodTower(X,T) :- \\
&\quad block(X), state(T), \\
&\quad on(X, table, T), \\
&\quad goal\_on(X, table).
\end{aligned}
$$

$$
\begin{aligned}
&goodTower(X,T) :- \\
&\quad block(X),\ block(Y),\ state(T),\ X != Y, \\
&\quad on(X,Y,T), \\
&\quad goal\_on(X,Y), \\
&\quad goodTower(Y,T).
\end{aligned}
$$

The formal expression for the knowledge "move a block on top of a good tower" is the following rule:

#blocks	plan length	time(Second)	
		without	$SRK02$
16	8	3.20	1.09
17	9	2424.42	1.75
18	10	3146.81	2.58
19	10	17.500	2.050
20	10	22.760	1.560
21	11	more than 2 days	2.100
22	12	more than 2 days	4.160
23	12	more than 2 days	3.210
24	13	more than 2 days	4.500
25	14	more than 2 days	8.800
26	15	more than 2 days	17.890

Table 10: Experimental results of applying $SRK02$ to the blocks world problem

$SRK02$ $\quad IF\ not\ goodTower(Y,T)\ THEN\ not\ move(X,Y,T)$

In the original encoding of the blocks world problem, suppose the action rule for *move* is

$$
\begin{aligned}
&move(X,Y,T) :- \\
&\quad state(T),\ block(X),\ object(Y), \\
&\quad X\ !=Y, \\
&\quad not\ covered(X,Y), \\
&\quad not\ covered(Y,T), \\
&\quad not\ blockedMove(X,Y,T).
\end{aligned}
$$

After applying $SRK02$, the *action rule* becomes

$$
\begin{aligned}
&move(X,Y,T) :- \\
&\quad state(T),\ block(X),\ object(Y), \\
&\quad X\ !=Y, \\
&\quad not\ covered(X,Y), \\
&\quad not\ covered(Y,T), \\
&\quad not\ blockedMove(X,Y,T), \\
&\quad \texttt{goodTower}(Y,T).
\end{aligned}
$$

Table 10 shows the experimental results before and after applying $SRK02$ in the blocks world planning domains. From Table 10 we see that all the instances could be solved in just 20 seconds. For the hard instances, which take the original encoding more than 2 days, while after applying rule $SRK\ II$, these instances become not that difficult.

Applying SRK to the logistics problem

In the logistics problem, "*do not return a package to its location*" is state relationship knowledge. This rule could be represented using the following formalized expression:

#package	steps	Time(Seconds)	
		without	SRK03
7	9	2.16	1.27
6	9	1.88	1.24
5	11	3.16	2.29
4	10	70.28	51.71
10	9	more than 2 days	41.26

Table 11: Experimental results of applying $SRK03$ to the logistics problem

$SRK03$ $(\ at(Package, Loc, T), not\ at(Package, Loc, T+1)\ ),$
$(at(Package, Loc, T_2), T_2 > (T+1)\ )\ CANNOT\ COEXIST$

This rule talks about the relationship when a package is returned to a location. The relationship is three-state-related: at state T, the package is at the location Loc, which could be the initial location of the package, or any other location. At the next state $T+1$, the package is not at location Loc, which means has been picked up by a truck or an airplane. At a later state T_2, the package stays at location Loc again. To prevent this relationship from happening can prevent at least two sets of actions. One is that once a package is picked up by a truck or an airplane, it is moved immediately. The action sequence $\{load\_truck(Pack, Tr, T)$, $unload\_truck(Pack, Tr, T+1)\}$ (or $\{load\_plane(Pack, P, T)$, $unload\_plane(Pack, P, T+1)\}$), will not happen. The other case is that once the package is actually moved, do not move it back to this location later. If the relationship exists, all the actions from picking it up to moving it around, until putting it down, make up a set of redundant actions.

The encoding of knowledge rule $SRK03$ is a constraint as follows:

$\leftarrow\ at(Package, Loc, T), not\ at(Package, Loc, T_1),$
$at(Package, Loc, T_2), T_1 = T+1, T_2 > T_1.$

Table 11 shows the experimental results before and after applying $SRK03$ in the logistics problem domains. For all the instances in Table 11, the original encodings of the logistics domains need more time to get an optimal plan than the ones using this SRK rule.

Applying SRK to the gripper problem

Similar knowledge could apply to the gripper problem:*"do not return a ball to a room."* This rule can prevent redundant action sequences, such as $\{pickup(G, A, T)$, $putdown(G, A, T+1)\}$ which provides no effects to ball A. Also, "*moving a ball to another room then moving it back*" will not happen if this rule is applied. The rule could be expressed formally as follows:

$SRK04$ $(at(A, R, T), not\ at(A, R, T_1), T_1 = T+1)$
$and\ (at(A, R, T_2), T_2 > T+1)\ CANNOT\ COEXIST$

The encoding of the knowledge is the following constraint

#ball	Time(Seconds)	
	without	SRK04
2	0.02	*0.02*
4	0.780	*0.12*
5	136.07	5.76
6	2611.740	21.23
7	more than 2 days	788.41
8	more than 2 days	3477.53

Table 12: Experimental results of the gripper problem with/without SRK

instance	#car	#location	steps	Time(Seconds)	
				without	SRK05
p03	4	4	11	1.04	0.84
p05ss	5	4	12	3.09	2.15
p04	5	4	15	124.44	40.15
p05	6	6	15	172.22	64.810
p04s	4	4	16	93.67	61.04

Table 13: Experimental results of the ferry problem with/without SRK (have solution)

$$\leftarrow at(A,R,T),\ not\ at(A,R,T_1), at(A,R,T_2),\\ T_1 = T+1, T_2 > T_1.$$

Table 12 compares the search time for each instance before and after applying the SRK rule $SRK04$, and shows that this rule can improve the search for the gripper domains. The overhead of applying the knowledge is very small. No new predicate is introduced into the original encoding; only new constraints are added. These new constraints are able to prune redundant actions existing in all instances which is proved by the data in Table 12.

Applying SRK to the ferry problem

For the ferry planning problem, *"do not return a car to a location"* is also applicable. Similarly, the knowledge is formally represented as the following rule:

$SRK05$ $(at(Car,Loc,T), not\ at(Car,Loc,T_1), T_1 = T+1)$
and $(at(Car,Loc,T_2), T_2 > T+1)\ CANNOT\ COEXIST$

Table 13 shows the empirical results.

Applying SRK to the elevator problem

For the elevator planning problem, *"do not return a passenger to a floor"* is also applicable. Similarly, the knowledge is formally represented as the following rule:

$SRK06$ $(at(P,F,T), not\ at(P,F,T_1), T_1 = T+1)$
and $(at(P,F,T_2), T_2 > T+1)\ CANNOT\ COEXIST$

#passenger	#floor	steps	Time(Seconds)	
			without	$SRK06$
1	3	3	0.00	0.00
2	4	6	0.00	0.00
3	5	10	2.14	1.75
4	7	12	73.01	50.43
5	9	14	1452.90	608.79
6	11	14	3676.65	890.52

Table 14: Experimental results of the elevator problem with/without SRK

Table 13 shows the empirical results.

Summary of SRK

From the above experimental data, we can summarize as follows:

1. SRK can be extracted from any planning domain.
2. SRK can improve the search efficiency for planning.
3. Sometime, SRK will introduce new predicates into the program, therefore, they may give rise to some extent of overhead. However, since the speedup is so strong that the benefit is far stronger than the overhead, the performance is improved significantly. Especially in the blocks world domain which is said to be notoriously hard, the hard instances that could not be solved in more than 2 days can be solved in ONE minute after applying SRK rules.

6.4 Experiments in procedural knowledge

Applying PK to the gripper problem

For the gripper domain, we test several knowledge rules to demonstrate how procedural knowledge prunes search space.

One is to apply procedural knowledge "*a gripper must pick up a ball before putting it down*". This PK rule is formally represented as

$PK01$ $\quad pickup(G, A, T_1) \Rightarrow putdown(G, A, T_2)$

which is encoded as an extra constraint added to the original encoding of the gripper problem:

$$: -pickup(G, A, T_1), putdown(G, A, T_2), T_2 < T_1.$$

This rule extracts sequential knowledge between two actions, *pickup* and *putdown*, for the same ball A. It provides dependency information about these two actions. If there is an action $pickup(G, A, T_1)$ in the plan, $putdown(G, A, T_2)$ where

$T_2 \leq T_1$ should not be in the plan. Therefore, all the atoms instantiated from $putdown(G, A, T_2)$ where $T_2 \leq T_1$, will get value $false$ immediately, and will not be choice points. The search space resulting from these atoms is pruned. Similarly, if $putdown(G, A, T_2)$ gets value $true$ first, all the atoms instantiated from $pickup(G, A, T_1)$ where $T_2 \leq T_1$ will get value $false$, and the search space from these atoms will also be pruned. As a result, all the "bad" plans with the sequence $\{putdown(left, ball\_a, 1), pickup(left, ball\_a, 2)\}$ will be pruned.

Another test is to apply PK with known state distance. Procedural knowledge "*if a gripper picks up a ball at state T, it must put down the ball at state T_2 where $T_2 - T = 2$*" is formalized using rule $PK02$:

$$PK02 \quad : pickup(G, A, T) \prec_2 putdown(G, A, T_2)$$

which can be encoded as follows:

$$\begin{aligned} &pickup(G, A, T) : - \\ &\quad putdown(G, A, T_2), \quad (5.9) \\ &\quad T_2 - T = 2. \\ &putdown(G, A, T_2) : - \\ &\quad pickup(G, A, T) \quad (5.10) \\ &\quad T_2 - T = 2. \end{aligned}$$

In the original encoding of the gripper problem, action rules for *pickup* and *putdown* are encoded as:

$$\begin{aligned} &\{pickup(G, A, T)\} : - \\ &\quad free(G, T), \\ &\quad at(robot, R_1, T), \\ &\quad at(A, R_1, T), \\ &\quad not\ affected(robot, T), \\ &\quad not\ goal(T). \\ &\{putdown(G, A, T_2)\} : - \\ &\quad at(robot, R, T), \\ &\quad not\ affected(robot, T), \\ &\quad on(A, G, T), \\ &\quad not\ goal(T). \end{aligned}$$

Before applying the procedural knowledge, $pickup(G, A, T)$ and $putdown(G, A, T)$ have no direct relationship with each other. To search the value of $pickup(G, A, T)$ and $putdown(G, A, T)$, we need to search the values of their preconditions. Once the procedural knowledge expressed as (5.9) and (5.10) is added to the encoding of the gripper problem, these two predicates are directly dependent. From (5.9) we know that if $putdown(G, A, T_2)$ is *true* in a stable model, $pickup(G, A, T)$ must be *true* in the model. From (5.10) we get that if $pickup(G, A, T)$ is *true* in the stable model, so is $putdown(G, A, T_2)$. Therefore, we do not need to search the values of the preconditions for these two actions and the search space that is used to search the values of preconditions is pruned, as shown in Figure 8.

There are other two PK rules. Rule $PK03$ expresses the knowledge that " *if the robot is at a room R, then its movement from room R to another room must occur*

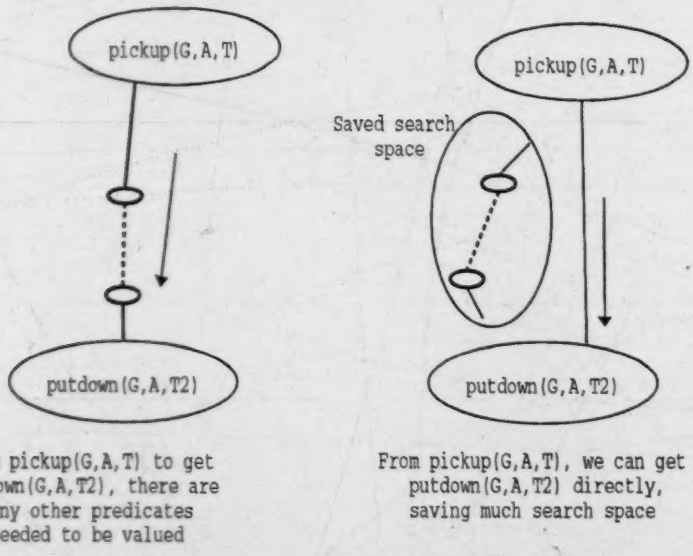

pickup(G,A,T)
putdown(G,A,T2)
From pickup(G,A,T) to get putdown(G,A,T2), there are many other predicates needed to be valued
pickup(G,A,T)
Saved search space
putdown(G,A,T2)
From pickup(G,A,T), we can get putdown(G,A,T2) directly, saving much search space

Figure 8: Procedural knowledge prunes search space

#ball	steps	Time(Seconds)					
		without	PK01	PK02	PK03	PK04	PK01+PK02
2	2	0.01	0.00	0.00	0.01	0.00	0.00
4	6	0.78	0.53	0.18	0.09	0.21	0.19
5	11	136.07	46.33	65.40	23.87	41.07	12.55
6	11	2611.74	440.77	148.34	177.41	81.00	23.74
7	15	> 2 days	47132.46	34631.98	38435.04	11550.75	7245.33

Table 15: Experimental results of the gripper problem with/without PK (have solution)

after a pickup or putdown action; if there is a pickup or putdown action, a move action must follow." $PK04$ formalizes the sequence to move ONE ball from a room to another room, using a robot: "*pickup the ball first, then move to another room, and put down the ball*".

$PK03$ $\quad at(robot, R, T) : (\ pickup(G, A, T) \mid putdown(G, A, T)\ ) \prec_1 move(robot, R, R_1, T_1)$

$PK04$ $\quad at(robot, R, T) : pickup(G, A, T) \prec_1 move(robot, R, R_1, T_1) \prec_1 putdown(G, A, T_2)$

$PK02$ is stricter than $PK01$ because when applying $PK02$, once one action is added to a plan, another one must also be added to the plan, and all the search to get the second action to the plan is pruned. However, this rule does not prune all the actions that $PK01$ could prune. Similarly, $PK01$ cannot prune all the bad actions that $PK02$ could prune. Applying these two rules together, therefore, will help prune more search space.

#ball	steps	Time(Seconds)				
		without	PK01	PK02	PK03	PK04
2	2	0.01	0.00	0.00	0.02	0.01
4	6	0.28	0.21	0.08	0.10	0.24
5	10	372.99	96.29	67.78	60.32	172.09
6	10	973.72	178.41	148.92	132.45	345.22

Table 16: Experimental results of the gripper problem with/without PK (no solution)

Table 15 demonstrates the performance of the encoding with and without procedural knowledge, when the given length of plan is long enough to get an optimal plan. Table 16 shows the performance with and without procedural knowledge when the given length of plan is too short to get a solution. The experiments show that all procedural knowledge can improve the search efficiency. $PK01$ and $PK02$ can compensate each other, thus the performance of the search incorporating these two rules is better than the search which only incorporates one of them.

Applying PK to the ferry problem

Rule $PK05$ formalizes the knowledge "*unboarding a car must happen 2 states later than boarding that car in a different location* ." This procedural knowledge rule involves the unknown objects, *location* since the information of where to board and unboard the car is not provided by the knowledge.

$PK05$ $\quad (Loc \neq Loc_1, dis(T_1, T) = 2):$
$board(Car, Loc, T) \mapsto unboard(Car, Loc1, T_1)$

Rule $PK05$ is encoded as following constraint:

$$\leftarrow board(Car, Loc, T), unboard(Car, Loc_1, T_1), Loc\,! = Loc_1, T_2 - T \neq 2.$$

Rule $PK06$ represents the partial procedural knowledge"*if ferry board Car at state T, then the ferry should move to this car's destination.*"

$PK06$ $\quad goalAt(Car, Loc_1):$
$board(Car, Loc, T) \preceq_1 move(ferry, Loc, Loc_1, T_1)$

Rule $PK06$ is encoded as the following inference rule:

$move(ferry, Loc, Loc_1, T_1) \leftarrow$
$\quad board(Car, Loc, T),$
$\quad goal\_at(Car, Loc_1),$
$\quad T_1 = T + 1.$

No matter whether the instance is solvable or not, the procedural knowledge helps improve the search performance. Table 17 and Table 18 demonstrate the performance of search before and after applying the procedural knowledge.

instance	#car	#location	steps	Time(Seconds)		
				without	PK05	PK06
p03n	4	4	9	0.16	0.21	0.12
p04n	5	4	12	5.49	1.78	0.82
p05n	5	4	12	8.72	4.18	4.05
p03	4	4	11	1.83	1.00	0.30
p04	5	4	15	65.18	6.70	2.66
p05	6	6	15	175.22	15.40	108.48

Table 17: Experiment results of the ferry problem with/without PK (have solution)

instance	#car	#location	steps	Time(Seconds)		
				without	PK05	PK06
p03n	4	4	8	0.04	0.19	0.05
p04n	5	4	11	1.77	1.06	0.56
p05n	5	4	11	9.06	3.54	7.19
p03	4	4	10	0.69	0.45	0.47
p04	5	4	14	59.17	7.42	21.38
p05	6	6	14	399.47.67	31.05	200.76

Table 18: Experimental results of the ferry problem with/without PK (no solution)

Applying PK to the logistics problem

$PK07$ represents "*truck should load a package, then in some later state unload it*". This prevents the possibility that action $unload\_truck(Package, Tr, T_2)$ is in a plan, while action $load\_truck(Package, Tr, T_1)$ where $T_1 > T_2$ takes place in the same plan.

$PK07 \qquad load\_truck(Package, Tr, T_1) \Rightarrow unload\_truck(Package, Tr, T_2)$

Table 19 shows the search performance carried out to get the first optimal plan before and after applying this PK rule. This table clearly tells us that the procedural knowledge applied here is helpful.

This kind of procedural rule is especially useful for planning problems encoded in logic programming since when a logic programming solver performs the search, all the atoms in all the states have the same chance to be valued first. The solver does not assign values to atoms at the earlier state first and search values for atoms at later state later. Therefore, it may assign values for atoms representing conditions or actions for some intermediate state first, then assign values for the atoms representing the conditions or actions for the previous states. This procedure of assigning value to atoms ensures that the situation that "a package is unloaded before it has ever been loaded" is valid during the search until later on when conflicts arise. Using procedural knowledge such as $PK07$ will prevent this from happening. If $unload\_truck(Package, Tr, T_2)$ is assigned *true* and $load\_truck(Package, Tr, T_1)$ also gets *true* where $T_1 > T_2$, $PK07$ will provide a conflict to stop further search based on this wrong assignment.

instance	#package	steps	Time(Seconds)	
			without	PK07
p04	7	9	2.11	2.06
p01	6	9	1.88	1.63
p02	5	7	3.16	2.90
p05	4	11	70.28	60.86

Table 19: Experimental results of the logistics problem with/without PK (have solution)

#passenger	#floor	steps	Time(Seconds)	
			without	PK
1	3	3	0.00	0.00
2	4	6	0.00	0.00
3	5	10	2.14	0.75
4	7	12	73.01	11.48
5	9	14	2342.04	115.33
6	11	14	4404.35	247.11

Table 20: Experimental results of the elevator problem with/without PK (have solution)

Applying PK to the elevator problem

For the elevator planning problem, *"Do not stop at a floor if the lift does not board or unboard passengers at next state"* is an applicable procedural knowledge. The knowledge can be formally represented as the following rules:

$PK08$ $\quad up(lift, From, To, T) \mapsto_1 down(lift, To, F, T_1)$.

$PK08$ $\quad up(lift, From, To, T) \mapsto_1 up(lift, To, F, T_1)$.

$PK08$ $\quad down(lift, From, To, T) \mapsto_1 up(lift, To, F, T_1)$.

$PK08$ $\quad down(lift, From, To, T) \mapsto_1 down(lift, To, F, T_1)$.

Table 20 and Table 21 show the empirical results.

Summary of utilizing procedural knowledge

For some planning domains, it is hard to extract sequential patterns from the final plans, such as the blocks world domain. For some other planning domains, the final plans provide clearly sequential information. For example, the gripper domain and the ferry domain.

However, once the procedural knowledge is extracted and encoded using our method, the search performance will be improved. Procedural knowledge can prune "bad" plans no matter whether the underlying instance has a solution or not. The encoded procedural knowledge can directly derive new actions based on the existing actions and it can also work as an extra constraint to prevent bad actions from being in the final plans.

#passenger	#floor	steps	Time(Seconds)	
			without	PK
1	3	2	0.00	0.00
2	4	5	0.00	0.00
3	5	9	0.87	0.75
4	7	11	34.40	33.78
5	9	13	2535.98	1273.54
6	11	13	3887.47	3656.65

Table 21: Experimental results of the elevator problem with/without PK (no solution)

6.5 Experiments in symmetry knowledge

Not every planning domain contains symmetry knowledge, especially when parallel actions are allowed. To encode symmetric knowledge, we have to introduce new predicates into the program. If the symmetry in the planning domain is not strong, the increased search space from these newly introduced predicates may outweigh the search space pruned by the symmetric knowledge. In this case, the symmetry knowledge may not improve the search performance. Whether symmetry knowledge works or not depends on the individual planning domain.

Applying SK to the gripper problem

The gripper planning domain has many symmetries. Since all the balls that are not moved by the robot stay at the "beginning" room and they all have the same goal room, these balls are symmetric objects for action *pickup*. When a gripper tries to pick up a ball, any ball in the room is a valid choice. Therefore, applying symmetry knowledge in the gripper domains will definitely help improve the search performance.

Since we identify balls by number, i.e., ball 1 and ball 2, so the order of the ball is the name of the ball. This is expressed by the following rule:

$$order(ball, A, A) \leftarrow ball(A).$$

We construct the following rule to identify symmetric object *ball*:

$$\begin{aligned} &symmetric(ball, A, B, T) \leftarrow \\ &\quad at(A, R, T), \\ &\quad at(B, R, T), \\ &\quad A \mathrel{!}= \mathbf{B}, \\ &\quad begin\_room(R). \end{aligned}$$

We construct the following rules to allow at most two balls that can be picked up by grippers. Predicate $least(ball, 2, A, T)$ means that ball A is one of the two balls that are allowed to be pick up.

$$
\begin{aligned}
&greater(ball, A, B, T) \leftarrow \\
&\quad symmetric(ball, A, B, T), \\
&\quad order(listName, A, N_a), \\
&\quad order(listName, B, N_b), \\
&\quad N_b > N_a.
\end{aligned}
$$

$$
\begin{aligned}
&nonLess(ball, 1, A, T) \leftarrow \\
&\quad symmetric(ball, A, B, T), \\
&\quad greater(ball, A, B, T), \\
&\quad A \mathrel{!}= B.
\end{aligned}
$$

$$
\begin{aligned}
&least(ball, 1, A, T) \leftarrow \\
&\quad not\ nonLess(ball, 1, A, T), \\
&\quad symmetric(ball, A, B, T).
\end{aligned}
$$

$$
\begin{aligned}
&nonLess(ball, 2, A, T) \leftarrow \\
&\quad symmetric(ball, A, C, T), \\
&\quad not\ least(ball, 1, A, T), \\
&\quad not\ least(ball, 1, C, T), \\
&\quad greater(ball, A, C, T), \\
&\quad C \mathrel{!}= A.
\end{aligned}
$$

$$
\begin{aligned}
&least(ball, 2, A, T) \leftarrow \\
&\quad not\ nonLess(ball, 2, A, T). \\
&\quad not\ least(ball, 1, A, T).
\end{aligned}
$$

$$
\begin{aligned}
&least(ball, 2, A, T) \leftarrow \\
&\quad least(ball, 1, A, T).
\end{aligned}
$$

We know that the action *pickup* is the action that works on the symmetric *balls*, so in order to break symmetries, we change the action rule for *pickup* to allow it only to work on two balls at a state. The action rule becomes

$$
\begin{aligned}
&\{pickup(G, A, T)\} \leftarrow \\
&\quad free(G, T), \\
&\quad at(A, R, T), \\
&\quad at(robot, R, T), \\
&\quad not\ affected(robot, T), \\
&\quad not\ goal(T), \\
&\quad \texttt{least}(ball, 2, A, T).
\end{aligned}
$$

Table 22 shows the time used to get the first optimal solution with and without symmetry knowledge for the gripper domain. When encoding symmetry knowledge into the program, new predicates are added. Since the symmetries are strong, the search space pruned outweighs the search space introduced by the new predicates.

Table 23 shows how well the symmetry knowledge improves the search when there is no solution for the gripper domain. When there is no solution, the symmetry knowledge prevents the underlying solver from trying symmetric solutions, and thus prunes search space. When there is no solution, the speedup is more impressive since all the symmetric branches are pruned.

#ball	steps	Time(Seconds)	
		without	SK
2	2	0.01	0.00
4	6	0.78	0.06
5	11	136.07	0.88
6	11	2611.74	9.53
7	15	more than 2 days	298.39
8	15	more than 2 days	1698.91

Table 22: Experimental results of the gripper problem with/without SK (have solution)

#ball	steps	Time(Seconds)	
		without	SK
2	2	0.02	0.10
4	6	0.28	0.05
5	10	372.99	2.73
6	10	937.72	3.86
7	14	more than 2 days	558.52
8	14	more than 2 days	762.20

Table 23: Experimental results of the gripper problem with/without SK (no solution)

Whether or not there is a solution, the speedup is up to 3 order of magnitude.

Applying SK to the ferry problem

In the ferry planning domains, whether there are symmetries depends on the instance. If, at the initial state, many cars stay at the same location, these cars are symmetric objects for action *board*, whether they have the same goal location or not. However, if there is not many such cars, the symmetry is not strong. Table 24 shows the initial conditions of the instances used in the experiments. In order to test whether the encoded symmetry knowledge could improve the search, we deliberately design 3 instances: p04s, p05ss, p05s. These instances have symmetric objects: *cars*. Instance p03 and p04 have symmetric objects, but not as strong as those of the instances p04s, p05s and p05ss.

Table 25 shows the search performances to get the first solution before and after applying the symmetry knowledge, when all the instances have solutions. Table 26 shows the search performances before and after applying the symmetry knowledge, when none of the instances have solution. Whether there is a solution or not, the search performance is improved by the symmetry knowledge. For instances p03n, p04n, and p05n, there is no symmetry in these instances, so applying symmetry knowledge will increase the search space because of the newly introduced predicates, and thus it will deteriorate the search performance.

Applying SK to the logistics problem

Instance	Initial state	Goal state
p03	at(car1,1,0). at(car2,1,0). at(car3,3,0). at(car4,3,0).	goalAt(car1,2). goalAt(car2,3). goalAt(car3,4). goalAt(car4,1).
p04	at(car1,1,0). at(car2,1,0). at(car3,3,0).at(car4,3,0). at(car5,3,0).	goalAt(car1,2). goalAt(car2,3). goalAt(car3,4). goalAt(car4,1). goalAt(car5,4).
p05ss	at(car1,3,0). at(car2,3,0). at(car3,3,0). at(car4,1,0). at(car5,1,0)	goalAt(car1,1). goalAt(car2,1). goalAt(car3,1). goalAt(car4,3). goalAt(car4,3).
p04s	at(car1,3,0). at(car2,3,0). at(car3,3,0). at(car4,3,0).	goalAt(car1,1). goalAt(car2,1). goalAt(car3,1). goalAt(car4,1).
p05s	at(car1,1,0). at(car2,1,0). at(car3,1,0). at(car4,1,0). at(car5,1,0). at(car6,1,0).	goalAt(car1,2). goalAt(car2,2). goalAt(car3,2). goalAt(car4,2). goalAt(car5,2). goalAt(car6,2).
p05n	at(car1,1,0). at(car2,2,0). at(car3,3,0). at(car4,3,0). at(car5,4,0).	goalAt(car1,2). goalAt(car2,3). goalAt(car3,4). goalAt(car4,1). goalAt(car5,6).
p04n	at(car1,1,0). at(car2,2,0). at(car3,3,0). at(car4,3,0). at(car5,4,0).	goalAt(car1,2). goalAt(car2,3). goalAt(car3,4). goalAt(car4,1). goalAt(car5,2).
p03n	at(car1,1,0). at(car2,2,0). at(car3,3,0). at(car4,4,0).	goalAt(car1,2). goalAt(car2,3). goalAt(car3,4). goalAt(car4,2).

Table 24: Instances of the ferry domains

instance	#car	#location	steps	Time(Seconds)	
				without	SK
p03	4	4	11	0.83	0.63
p04	5	4	15	65.18	45.10
p05ss	5	6	13	1.65	1.42
p04s	4	6	16	79.58	8.58
p05n	5	4	12	15.36	16.86
p04n	5	4	12	3.06	4.01
p03n	4	4	9	0.10	0.14

Table 25: Experimental results of the ferry problem with/without SK (have solution)

instance	#car	#location	steps	Time(Seconds)	
				without	SK
p03	4	4	10	0.41	0.28
p04	5	4	14	36.81	8.12
p05ss	5	4	11	1.92	0.92
p04s	4	4	15	93.67	7.43
p05s	6	6	14	more than 2 days	70.83
p05n	5	4	11	3.71	4.04
p04n	5	4	11	1.57	1.33
p03n	4	4	10	0.05	0.07

Table 26: Experimental results of the ferry problem with/without SK (no solution)

instance	#package	steps	Time(Seconds)	
			without	SK
p04	7	9	2.23	8.00
p01	6	9	1.62	4.83
p02	5	7	2.88	12.73
p05s	4	11	76.59	63.52
p04s	7	9	3.07	2.46
p05ss	5	11	35.99	45.56

Table 27: Experimental results of applying symmetry knowledge to the logistics problem (have solution)

For logistics problem, we test five instances, of which, p04s and p05s constructed to have symmetric trucks and planes at the initial state, while other three instances do not have any symmetric objects at the initial state.

Table 27 and Table 28 show the experimental results before and after applying symmetric knowledge.

Summary of utilizing symmetry knowledge

Not every planning domain has symmetries. In the blocks world planning domain, since we allow parallel actions, there is no symmetry left. Changing the order of

instance	#package	steps	Time(Seconds)	
			without	SK
p04	7	8	0.61	3.70
p01	6	8	0.32	1.62
p02	5	6	0.78	3.38
p05s	4	10	31.35	43.49
p04s	7	8	0.66	2.01
p05ss	5	10	17.78	38.20

Table 28: Experimental results of applying symmetry knowledge to the logistics problem (no solution)

action $move(a, b)$ will totally change the plan. Also, any two blocks that have the same properties must be at the *table* and be clear. Then, these two blocks can be moved at the same state. They are not symmetric objects.

If the planning domain has symmetric objects, identifying and breaking the symmetries, using the encodings we provide in Section 5.5, can prune search space. Especially when there is no solution, the encoding without symmetry knowledge will try all the symmetric branches; while after adding symmetry knowledge, only one of the symmetric branches is tried.

However, when symmetry is not strong, the new predicates that are used to identify and break symmetries may increase search space, sometimes this overhead can outweigh the benefit from pruning the symmetries. In this case, we do not apply symmetry knowledge. For example, in the logistics domain, only when at one state T, two trucks (or two airplane) have not loaded any packages, and they are at the same location, can these two trucks (or airplanes) be symmetric objects. This case is very rare in the logistics planning domain, so our predication is that the symmetries are not strong enough, it is of no benefit to break them, and the experiments proved that our prediction is right.

6.6 Experiments in distance knowledge

To encode distance knowledge is not a simple job since, in different situations, the formula to compute distance is different. Therefore, it is necessary to enumerate all possible situations. Also, in each situation, complete information about the current state is needed. At last, new choice points may be introduced because of the newly constructed predicates that are necessary to describe the properties of a state. So the overhead to encode distance knowledge may outweigh the benefits of applying it.

Applying DK to the gripper domain

For the gripper problem, we compute the lower bounds for all the balls. The distance is used to compute the minimum steps needed to move all the balls in the initial room to the goal room. Figure 9 shows the encoding of distance knowledge for the gripper domain.

To construct distance knowledge, we use predicate $inRoom(R, N, T)$ to represent that in state T there are N balls in room R. This predicate works as follows: only when the robot puts down the balls at a room, say R_1, does the number of balls in room R_1 increase, and the number of balls in another room decrease by the same number. Otherwise, the number does not change. Predicate $put(N, R, T)$ means that the robot puts down N balls in room R at state T.

Let N be the number of balls in the initial room at state T, then the formula is $(N+1)/2*4-1$, if the robot is at the initial room and has not picked up any balls. The formula is $(N+1)/2*4-2$ if the robot is moving from the initial room to the goal room. The formula is $(N+1)/2*4-3$ if the robot is at the goal room and has not put down any ball. The formula is $(N+1)/2*4$ if the robot is moving from the goal room to the initial room, since in this case, $N = N-2$.

```
inRoom(R,N+1,T+1)←
  put(1,R,T), inRoom(R,N,T).
inRoom(R,N+2,T+1):-
       put(2,R,T), inRoom(R,N,T).
put(2,R,T+1)←
       at(robot,R,T), putdown(G,A,T),
       putdown(G1,B,T), A != B, G != G1.
put(1,R,T+1)←
       at(robot,R,T), putdown(G,A,T),
       not put(2,R,T+1).
inRoom(R,N,T)←
       inRoom(R1,N1,T),
       N = balls - N1, R != R1.
f ← at(robot,R,T), begin_room(R),
    inRoom(R,N,T), num_gripper(2),
    pickup(G,A,T), R != R1,
    (((N+1) / 2) * 4 - 1) > (steps - T).
f ← at(robot,R,T), begin_room(R),
    inRoom(R,N,T), num_gripper(2),
    move(robot,R,R1,T), R != R1,
    (((N+1) / 2) * 4 - 2) > (steps - T).
f ← at(robot,R1,T), begin_room(R),
    inRoom(R,N,T), num_gripper(2),
    putdown(G,A,T), R != R1,
    (((N+1) / 2) * 4 - 3) > (steps - T).
f ← at(robot,R1,T), begin_room(R),
   in_room(R,N,T), num_gripper(2),
    move(robot,R1,R,T), R != R1,
    (((N+1) / 2) * 4 ) > (steps - T).
← f.
```

Figure 9: Construction of distance knowledge for the gripper problem

#ball	steps	time(Seconds)	
		without	DK
2	3	0.01	0.05
4	7	0.78	0.13
5	11	136.07	199.53
6	11	2611.74	549.63

Table 29: Experimental results of the gripper problem with/without DK (have solution)

#ball	steps	Time(Seconds)	
		without	DK
2	2	0.02	0.00
4	6	0.28	0.01
5	10	372.99	0.09
6	10	937.72	0.09
7	14	more than 2 days	0.16
8	14	more than 2 days	0.20

Table 30: Experimental results of the gripper problem with/without DK (no solution)

Table 29 shows the experimental results before and after applying distance knowledge for the gripper problem when searching for the first optimal plan. The overhead of distance knowledge is larger than the benefit it provides, so for small instances, the performance is worse. However, for hard instances, such as the instance with 6 balls, the benefit overcomes the overhead. For instances with more than 6 balls, it took both encodings more than 4 days, therefore we were not able to determine which one is better.

Table 30 shows the experimental results before and after applying distance knowledge to the gripper problem when there is no optimal plan. Since the lower bound is exactly the same as the length the for optimal plans, once the given length is less than the length of the optimal plans, our formula will get $f = false$, based on information at the initial state, so the search will stop immediately. That is why, for all the instances in Table 30, the encodings with distance knowledge will take less than one second to reach the condition that there is no plan.

Applying DK to the logistics domain

The following are the encodings used to compute lower bound in the logistics problem.

$$
\begin{aligned}
&low(Obj, 9, T) \leftarrow \\
&\quad not\ airport(Loc), \\
&\quad not\ airport(Loc_1), \\
&\quad at(Obj, Loc, T), \\
&\quad goalAt(Obj, Loc_1), \\
&\quad not\ sameCity(Loc, Loc_1), \\
&\quad Loc\ != Loc_1, \\
&\quad T > steps - 9.
\end{aligned}
$$

$$
\begin{aligned}
&low(Obj, 6, T) \leftarrow \\
&\quad airport(Loc), \\
&\quad not\ airport(Loc_1), \\
&\quad at(Obj, Loc, T), \\
&\quad goalAt(Obj, Loc_1), \\
&\quad not\ sameCity(Loc, Loc_1), \\
&\quad Loc\ != Loc_1,
\end{aligned}
$$

Instance	#package	steps	Time(Seconds)	
			without	DK
p01	6	9	2.34	2.33
p02	5	7	3.37	3.08
p04	7	9	3.41	1.98
p05	4	11	87.97	66.31
p07	10	9	more than 2 days	119.61

Table 31: Experimental results of applying DK to the logistics problem

$T > steps - 6.$

$low(Obj, 6, T) \leftarrow$
$not\ airport(Loc),$
$airport(Loc_1),$
$at(Obj, Loc, T),$
$goalAt(Obj, Loc_1),$
$not\ sameCity(Loc, Loc_1),$
$Loc\ != Loc_1,$
$T > steps - 6.$

$low(Obj, 3, T) \leftarrow$
$at(Obj, Loc, T),$
$goalAt(Obj, Loc_1),$
$sameCity(Loc, Loc_1),$
$Loc\ != Loc_1,$
$T > steps - 3.$

$f \leftarrow$
$low(Obj, N, T),$
$s(N),$
$T > steps - 9,$
$N > (steps - T).$

$\leftarrow f.$

The first inference rule computes the lower bound for a package in the conditions described in Figure 5. The lower bound in this case is 9. The second rule computes the lower bound, which is 6, for the packages whose current location is the airport and the goal location is a non-airport location in a different city. The third rule computes the lower bound, which is 6, for the packages whose goal location is an airport and the current location is a non-airport location in a different city. The forth rule computes the lower bound, which is 3, for packages whose current and goal locations are at the same city.

Table 31 shows the experimental results with and without applying the distance knowledge, when there is optimal solution. Table 32 shows the experimental results with and without applying the distance knowledge when there is no solution. Clearly, in both cases, the distance knowledge improves the search.

Applying DK to the blocks world domain

Instance	#package	steps	Time(Seconds)	
			without	DK
p01	6	8	0.67	0.53
p02	5	6	0.77	0.63
p04	7	8	0.94	0.65
p05	4	10	33.91	11.71
p07	10	8	4.47	3.48

Table 32: Experimental results of applying DK to the logistics problem

To encode distance knowledge into the encoding of Blocks World domain, we need to compute the lower bound that is needed to change the property of a block from the current state to the goal state. We use the following predicates to compute lower bound:

1. $disTop(X, N, T)$: there are N blocks on top of block X at state T
2. $finalTop(X, N)$: there are N blocks on top of block X at the goal state
3. $sameAsFinal(X, T)$: the current property of block X is the same as its goal property

Let X be a block, $BelowFinal$ be the block under block X at the goal state, and $BelowBlock$ be the block under block X at state T. Let $N_{disTop}(BelowFinal)$ be the N that makes $disTop(BelowFinal, N, T)$ *true*. $N_{finalTop}(X)$ is the N that makes $finalTop(X, N)$ *true*. We use three names to describe three different situations of a block:

1. CLEAR: A block is clear, not covered by another block
2. BLOCK: A block is between two blocks. It is not clear, it is not at the table.
3. TABLE: A block is at the table.

To compute distance for a block, we describe the following four scenarios:

1. CLEAR=>BLOCK

In this case, at the current state, a block is CLEAR. In the goal state, the block is BLOCK. To change the current property to its goal property, we need to move all the blocks above Y, then move X to Y, and then move all the blocks that are supposed to be above block X at the goal state to X.

The lower bound for block X at state T in the above situation is computed as following:

$$\texttt{lowerBound} = N_{disTop}(BelowFinal) + 1 + N_{finalTop}(X)$$

2. BLOCK=>BLOCK

In this case, for block X, both at the current state and the goal state, it is BLOCK. If the property of the block at the current state is the same as its goal property, do not compute its lower bound. Otherwise, the lower bound for block X at state T in the above three situations is computed as follows:

Instance	#blocks	steps	time(Second)	
			without	DK
bw.15	15	8	3.27	8.33
bw.17	17	9	2424.42	17.73
bw.18	18	10	3146.81	102.39
bw.19	19	10	more than 2 days	108.45
bw.20	20	10	22.760	102.23
bw.21	21	11	more that 2 days	112.47

Table 33: Experimental results of applying DK to the blocks world problem (have solution)

$$\texttt{lowerBound} = N_{disTop}(X) + 1 + N_{finalTop}(X)$$

3. BLOCK=>TABLE

4. TABLE=>BLOCK

In either of the above two cases, to change the current property to its goal property, we need to move all the blocks above X, then move X to Y, then move all the blocks that are supposed to be above block X at the goal state to X.

The lower bound for block X at state T in the above three situations is computed as follows:

$$\texttt{lowerBound} = N_{disTop}(X) + 1 + N_{finalTop}(X)$$

Table 33 shows the experimental results before and after applying the distance knowledge discussed above. When there is a solution, for small instances, such as *bw*.15 and *bw*.20, the overhead for applying distance knowledge outweighs the benefit. However, for hard instances, such as *bw*.17, *bw*.18, *bw*.21, the benefit of applying distance knowledge overcomes the overhead, and the search efficiency is improved significantly.

Table 34 shows the experimental results before and after applying distance knowledge when there is no solution for the instances. The table shows that when there is no solution for an instance, the distance knowledge does not improve the search performance; on the contrary, it slows down the search.

Applying DK to the ferry domain

In the ferry domain, we compute the lower bound for each car that is not at its goal location. There are three possible situations:

1. The car and the ferry are at the same location and the ferry is empty.
   The lower bound for a car in this case is 3. That is, at least three actions-*pickup*, *move* and *putdown*, are needed to move the car to its goal location. We apply this case only to the last three states.

Instance	#blocks	steps	time(Second)	
			without	DK
bw.15	15	7	0.34	1.18
bw.17	17	8	0.84	3.45
bw.18	18	9	1.05	6.69
bw.19	19	9	1.38	7.82
bw.20	20	9	1.36	5.53
bw.21	21	10	2.11	8.75

Table 34: Experimental results of applying DK to the blocks world problem (no solution)

instance	#car	#location	steps	Time(Seconds)	
				without	DK
p03	4	4	11	1.38	1.05
p04	5	4	15	159.18	146.54
p05	6	6	15	203.11	167.18
p03n	4	4	9	0.18	0.17
p04n	5	4	12	5.49	3.72
p05n	5	4	12	9.80	9.74

Table 35: Experimental results of the ferry problem with/without DK (have solution)

2. The car and the ferry are at different locations and the ferry is empty.
   The lower bound for a car in this case is 4. That is, at least four actions are needed to move the car to its goal location: *move* ferry to where the car is, *pickup* the car, *move* it to its goal location and *putdown* the car. We apply this case only to the last four states.

3. The car and the ferry are at different locations and the ferry is not empty
   The lower bound for a car in this case is 5. That is, at least five actions are needed to move the car to its goal location: empty the ferry (*putdown*), *move* ferry to where the car is, *pickup* the car, *move* it to its goal location and *putdown* the car. We apply this case only to the last five states.

Table 35 lists all the experimental results of searching the first optimal plan when the underlying planning domain has solutions. From the table, we can see that after applying the distance knowledge, the search performance is better, but not very impressive.

Table 36 shows the experimental results when there is no solution for the underlying planning domain. The distance knowledge make the search performance worse.

Applying DK to the elevator domain

In the elevator domain, we compute the lower bound for each passenger who has not his/her goal floor. There are three situations:

1. If the lift and the passenger are at the same floor, then it needs 3 steps to move the passenger to his/her goal floor: board, move and unboard.

instance	#car	#location	steps	Time(Seconds)	
				without	DK
p03	4	4	9	0.21	0.28
p04	5	4	14	59.89	65.92
p05	6	6	14	470.31	552.23
p03n	4	4	8	0.08	0.11
p04n	5	4	11	2.26	2.27
p05n	5	4	11	9.64	9.75

Table 36: Experimental results of the ferry problem with/without DK (no solution)

#passenger	#floor	steps	Time(Seconds)	
			without	DK
1	3	3	0.00	0.00
2	4	6	0.02	0.02
3	5	10	2.26	2.52
4	7	12	76.68	95.21
5	9	14	4032.22	3893.40

Table 37: Experimental results of the elevator problem with/without DK (have solution)

2. If the lift is not at the same floor as where the passenger is, it will need 4 steps: move to where the passenger is, board it, move it to the goal floor and finally, unboard it.

3. If the passenger is in the lift, it will need 2 steps to move it to its goal floor: move and unboard.

As the distance knowledge only works on the last few states, its overhead may outweigh its benefits. Table 37 and Table 38 the experimental results before and after applying distance knowledge to the elevator domain.

Summary of applying distance knowledge

Distance knowledge exists in all the planning domains, but may not improve search efficiency for every planning domain.

#passenger	#floor	steps	Time(Seconds)	
			without	DK
1	3	2	0.00	0.00
2	4	5	0.00	0.10
3	5	9	0.89	0.81
4	7	11	39.17	38.71
5	9	13	1496.85	1451.44

Table 38: Experimental results of the elevator problem with/without DK (no solution)

For the gripper domain, since the lower bound is used to compute the minimum steps to transfer all the balls that are not at their goal room to their goal room, at each state, only one lower bound is computed. The computation is simple; it does not need to check every ball's property. Also, the lower bound is exactly the length of optimal plans, using the distance knowledge only, we can tell if the given length can lead to any solution: if it is less than the lower bound at the initial state, then there will be no solution. In this case, all the search space is saved.

For the ferry domain, we cannot compute the minimum steps to move all the cars to their destinations because of the parallel actions. If a car's destination is another car's initial location, the ferry can put down the first car at its destination and board the second car at the same state. If we compute the lower bound for each car at every state, the overhead is too big. Before the lower bound is used to control the search, the properties of all the objects in the domain have to be checked in order to compute the lower bound. The effort required to do this is not much less than that required to backtrack without checking the lower bound, which is why distance knowledge in the ferry domain makes search performance worse. Also, in the ferry domain, since the distance knowledge only works at the last five states, it cannot prune much space.

For the logistics domain, although the distance knowledge works the same way as that in the ferry domain, the lower bound for some packages can be 9, which is comparable to the length of the final plans. The distance knowledge can therefore work at very early states which is why distance knowledge works better in the logistics domain than in the ferry domain.

In the blocks world domain, distance knowledge can work at all the states. The reason that distance knowledge works well for blocks world domain is the same as that of logistics domain.

For the elevator domain, the distance knowledge only works on the last 4 or less states. Therefore, the knowledge can not prune bad plans at early stage if the length of plan is much longer than 4. That is why the distance knowledge works not well at the elevator domain.

6.7 Summary of the experimental results

The above experimental results show that each class of domain dependent knowledge can prune some search space, but to what extent, depends on the planning domain and the type of knowledge.

End state knowledge, state relationship knowledge and procedural knowledge can prune bad actions for almost all the planning domains and the improvement can be up to 3 orders of magnitude.

Symmetry knowledge can improve search efficiency, but the overhead is a concern. When the planning domain has strong symmetry, symmetry knowledge will improve the search significantly. The experimental results of applying symmetry knowledge to the gripper domain proved this. When an instance has strong symmetries, the benefits can outweigh the overhead; otherwise, the encoding of the symmetry knowledge will increase the search space.

Distance knowledge can improve search efficiency, but not for every domain. As we discussed in Section 6.6, if the lower bound for all the objects in the domain can be computed, it can be used as a checker to stop an invalid partial plan at an early stage. If only the lower bound for each object in the domain can be computed, the improvement depends on the coverage of the distance knowledge. If the distance knowledge only works on the last few states, the encoding of the knowledge may introduce more overhead than benefit. If the distance knowledge works at almost all the states, the improvement can be significant, especially for hard instances.

Combining different classes of domain dependent knowledge can prune more search space. For example, we can add end state knowledge and symmetry knowledge to a logic program representing a planning problem. Since these two classes of knowledge prune different sets of redundant actions, the encoded knowledge can compensate each other. Take the gripper problem as an example. The end state knowledge can prune redundant actions for the balls that are already in the goal room. If the balls are in the goal room, then no actions need to be undertaking for these balls. While symmetry knowledge works on pruning bad actions for the balls that are not in the goal room. Therefore, combining these two classes of knowledge can prune more search space.

The utilizations of some knowledge may make other knowledge become useless. For example, in the same gripper problem. If we apply procedural knowledge to specify that the sequence "a gripper first picks up a ball, then moves it to the goal room, puts it down, and moves back to the beginning room" is in final plans. At the same time, we also apply end state knowledge saying that when a ball a and a gripper g are in the beginning room at any state T, action $putdown(g, a, T)$ should not be taken, and if the ball is in the goal room, action $pickup(g, a, T)$ should not be taken. Then, the state relationship knowledge " do not move a ball back to a room" becomes useless after applying the above two classes of knowledge.

7 Conclusion and Future Work

7.1 Conclusions

In this thesis, we have investigated the methodology to encode domain independent knowledge and domain dependent knowledge of planning problems in logic programming. Moreover, a classification for domain dependent knowledge is provided which can be further used as a guide to help users extract domain knowledge from a planning domain.

For each class of domain knowledge, a formal representation and a standard encoding are proposed. Examples have shown how standard encoding can translate the knowledge into constraints that can be added to logic programs representing the underlying planning domains in order to make planning more efficient.

The empirical evidence indicates that (1) domain dependent knowledge is available in most, if not all, planning domains; (2) the encoding provided in this thesis for each class of domain dependent knowledge can improve search efficiency under the logic programming context; and (3) domain dependent knowledge can improve search performance for planning problems encoded in logic programming.

7.2 Future work

More extensive experiments are needed. In this thesis, we conducted experiments on five benchmark problems, to test if our method works on all the planning problems, we want to explore more planning domains in the future.

The encoding of planning problem discussed in this thesis allows parallel actions. Therefore, all the representation and encodings of domain dependent knowledge are constructed in this context. The performance of some domain dependent knowledge will be different if no parallel actions are allowed. For example, when encoding distance knowledge for some planning domains, the concurrency of actions makes it hard to compute lower bound for all the objects. If no parallel actions are allowed, the lower bound for all the cars in the ferry domain can be computed easily, as we compute lower bound for balls in the gripper domain. More research needs to be done to investigate how domain dependent knowledge works on the encoding that does not allow parallel actions.

Based on the formalization and standard encoding of each class of domain knowledge, an important area for future research will be to design a system to automatically generate and translate domain specific knowledge to be added to the encoding of planning domains in logic programming.

A Smodels' Syntax

In this section, we briefly describe some terms in Smodels' language that will be used to encode the planning problems. For the language details, please refer to [36].

A.1 Constant,Variable and Literal

Smodels programs allow variables and constants. A variable in Smodels must start with an upper case. It can be a string of letters and numbers and it can contain underscores.

There are two ways to assign value to a constant. One way is to assign the value in a program such as:

$$const\ \ steps\ =\ 10$$

which says that, from that point on, every occurrence of the constant *steps* will be substituted by 10. Another method to achieve the same result is to use the $-c$ command line option, as shown here:

$$lparse\ \ -c\ \ steps = 10\ \ myProgam.sm \mid smodels$$

Smodels has a compact way to define numerical domains for variables: the use of *range*. A range is of the form:

$$start\ \ \ldots\ \ end$$

where *start* and *end* are numeric constants, or symbolic constants acting as numeric constants, or constant valued arithmetic expressions. For example, range $blocks(1 \ldots 3)$ represents the same facts as follows:

```
blocks(1). blocks(2). blocks(3).
```

A.2 Rules

An inference rule in Smodels is of the form:

$$head(Z_1, .., Z_k)\ \ :-\ \ a(X_1, .., X_n),\ \ not\ \ b(Y_1, ..., Y_m).$$

where $head(Z_1, ..., Z_k)$ is the head of the rule, and $a(X_1, .., X_n)$ and $not\ b(Y_1, ..., Y_m)$ make up the body. X_i, Y_j, Z_l are variables which are the parameters of predicates a, b, and *head*, respectively.

The Smodels system also includes choice rule, cardinality rule, and conditional literal discussed in Section 2.4.

A.3 Statements

There are several statements used in the Smodels system. We explain them briefly.

1. #domain

This statement is used to write down the domain predicates for variables compactly. The syntax is

$$\texttt{\#domain}\ \ domain_1(V_1),\ ...,\ domain_n(V_n).$$

When we write down domain-restricted rules in Smodels, if we use the above domain statement, when variable V_1 is needed in a rule, $domain_1(V_1)$ does not need to appear in the body of the rule.

2. #hide and #show

The *#hide* statement without any parameters (as shown below):

#hide.

tells Smodels not to print out the stable model. If a *#show* statement follows the above *#hide* statement with some parameters, it means that Smodels will print out all the predicates listed after *#show* that are *true* in the found stable model. For example,

#hide.
#show $move(X, Y, T)$.

The above statements tells the Smodels that if there is a stable model, print out all atoms from predicate $move(X, Y, T)$ which are *true* in the model.

3. *Compute*

This statement specifies what kind of stable models are needed. The syntax is

compute $n\{a_1, .., a_n\}$.

where n is the number of stable models, in which $a_1, .., a_n$ are *true*, and needed to search. If n is omitted, that means Smodels needs to find only one stable model that satisfies the specification.

B Encodings of the Blocks World Problem in Smodels

Encoding of instance : initial and goal states

```
% objects
block(a).block(b).block(c).block(d).
base(table).
base(X) :- block(X).

%initial state
on(a,b,0).on(b,table,0).on(c,table,0).

%goal predicate
goal(T):-
   state(T),
   on(b,c,T),
   on(c,table,T),
   on(a,table,T).

goal(T+1):-goal(T).
goal:-goal(T).
compute {goal}.
```

Encoding of action system

```
%Modified by Xiumei Jia from coding of Ilkka Niemela[25]

%state domain
state(0..steps).

#hide.
#show move(X,Y,T).

% action choice
move(X,Y,T):-
       state(T),block(X),base(Y),
       X != Y,
       not covered(X,T),
       not covered(Y,T),
       not affected(Y,T),
       not blockedMove(X,Y,T),
       not goal(T).

blocked_move(X,Y,T) :-
      state(T),block(X),base(Y), X != Y,
      not move(X,Y,T).

blocked_move(X,Y,T) :-
      block(X), base(Y),
      object(Z),state(T),
```

```
      X != Y,
      move(X,Z,T), X != Z,
      Y != Z.

blocked_move(X,Y,T) :-
      block(X), block(Y),
      block(Z), state(T),
      X != Y,Z != Y,
      move(Z,Y,T),
      X != Z.

%Identify effected objects

affected(X,T) :-
     state(T), block(X),base(Y),
     move(X,Y,T), X != Y.

% effects

on(X,Y,T+1) :-
      block(X),base(Y),
      move(X,Y,T),X != Y.

% frame axioms

on(X,Y,T+1) :-
      block(X), base(Y),
      on(X,Y,T), X != Y,
      not affected(X,T).

% a block is covered if there is another
% block on top of it

covered(X,T) :-
      block(Z), block(X),
      Z != X,
      on(Z,X,T).
```

C Encodings of the Logistics World in Smodels

Encoding of instance : initial and goal states

```
% objects
package(p1).package(p2).
plane(1). plane(2).
city(c1). city(c2).
location(c11). location(c12). location(c21). location(c22).
in_city(c11,c1). in_city(c12,c1).
in_city(c21,c2). in_city(c22,c2).
truck(11). truck(12). truck(21).
airport(c12). airport(c22).

%initial state
at(p1,c11,0). at( p2,c22,0).
at_p(1,c12,0). at_p(2,c12,0).
at_t(11,c11,0). at_t(12,c11,0). at_t(21,c21,0).

%goal predicate
goal(T):-
   state(T),
   at(p1,c22,T),
   at(p2,c22,T).

goal(T+1):-goal(T).
goal:-goal(T).
compute {goal}.
```

Encoding of action system

```
%Written by Xiumei Jia

% state domain
state(0..steps).

#hide.
#show load_truck(Obj, Tr, T),load_airplane(Obj, P, T).
#show unload_truck(Obj, Tr, T),unload_airplane(Obj, P, T).
#show drive_truck(Tr,From,To,T),fly_airplane(P,From,To,T).

#domain package(Obj),state(T),state(T1),state(J),truck(Tr).
#domain location(Loc),location(Loc1),location(Loc2).
#domain plane(P),location(From),location(To).
#domain container(Cont),city(C).


% Action choice

% Action:load_truck(Obj, Tr, T)
% Preconditions:
```

```
%  at(Obj,Loc, T)
%  at_t(Tr, Loc, I)

{ load_truck(Obj, Tr, T) } :-
     at(Obj, Loc, T),
     at_t(Tr, Loc, T),
     not goal(T),
     not affected_t(Tr,T).

% Action: load_airplane(Obj, P, T)
% Preconditions:
% at(Obj,Loc, T)
% at_p(P, Loc, T)
{ load_airplane(Obj, P, T) } :-
     at(Obj, Loc, T),
     at_p(P, Loc, T),
     not affected_p(P,T),
     not goal(T),
     airport(Loc).
% Action: unload_truck(Obj, Tr, T)
% Preconditions:
%  in(Obj, Tr, T)
%  at_t(Tr, Loc, T)

{ unload_truck(Obj, Tr, T) } :-
     in(Obj, Tr, T),
     not affected_t(Tr,T),
     not goal(T).

% Action: unload_plane(Obj, P, T)
% Preconditions:
% in(Obj, P, T)

{ unload_airplane(Obj, P, T) } :-
     in(Obj, P, T),
     not goal(T),
     not affected_p(P,T).

% Action: drive_truck(Tr, From,To, T)
% Preconditions:
% at_t(Tr, From, T)

{ drive_truck(Tr, From, To, T) } :-
     at_t(Tr, From, T),
     same_city(From, To),
     not goal(T),
     From != To.

% Action: fly_plane(P, From,To, T)
% Preconditions:
% at_p(P, From, T)

{ fly_airplane(P, From, To, T) } :-
```

```
      at_p(P, From, T),
      airport(From), airport(To),
      not goal(T),
      not same_city(From,To),
      From != To.

% Constraints:

:- 2 { at(Obj, Loc, T):location(Loc),
      in(Obj, Cont, T):container(Cont)},
      state(T),package(Obj).

:- 2 { at_t(Tr, Loc, T):location(Loc) },state(T), truck(Tr).

:- 2 { at_p(P, Loc, T):airport(Loc) },
     state(T),location(Loc), plane(P).

% Identify Affected Objects

affected(Obj, T) :-
     load_truck(Obj, Tr, T).

affected(Obj, T) :-
     load_airplane(Obj, P, T).

affected(Obj, T) :-
     unload_truck(Obj, Tr, T).

affected(Obj, T) :-
     unload_airplane(Obj, P, T).

affected(Tr, T) :-
     drive_truck(Tr, From, To, T),
     same_city(From, To),
     From != To.

affected(P, T) :-
     fly_airplane(P, From, To, T),
     airport(From), airport(To),
     From != To.

Effects:

in(Obj, Tr, T+1) :- load_truck(Obj, Tr, T).

in(Obj, P, T+1) :- load_airplane(Obj, P, T).

at(Obj, Loc, T+1) :- unload_truck(Obj, Tr, T), at_t(Tr, Loc, T).

at(Obj, Loc, T+1) :-
      unload_airplane(Obj, P, T),
      at_p(P, Loc, T), airport(Loc).

at_t(Tr, To, T+1) :-
```

```
      drive_truck(Tr, From, To, T),
      same_city(From, To), From != To.

at_p(P, To, T+1) :-
      fly_airplane(P, From, To, T),
      airport(From), airport(To),
      From != To.

% FRAME AXIOMS

at(Obj, Loc, T+1) :-
      at(Obj, Loc, T),
      not affected(Obj, T).

in(Obj, Cont, T+1) :-
      in(Obj, Cont, T),
      not affected(Obj, T),
      container(Cont).

at_t(Tr, Loc, T+1) :-
      at_t(Tr, Loc, T),
      not affected(Tr, T),
      truck(Tr).

at_p(P, Loc, T+1) :-
      at_p(P, Loc, T),
      not affected(P, T),
      plane(P), airport(Loc).

%Other predicates

same_city(Loc1, Loc2) :-
     in_city(Loc1, C),
     in_city(Loc2, C),
     Loc1 != Loc2.

container(X) :- truck(X).

container(X) :- plane(X).
```

D Encoding of the Ferry Problem in Smodels

Encoding of instance : initial and goal states

```
% objects
car(car).
location(1). location(2).

%initial state
at(car,1,0).
at(ferry,1,0).

%goal predicate
goal(T):-
   state(T),
   at(car,2,T).

goal(T+1):-goal(T).
goal:-goal(T).
compute {goal}.
```

Encoding of action system

```
%Written by Xiumei Jia

#domain state(T),car(Car),state(T2).
#domain location(Loc),location(From),location(To).

#hide.

#show board(Car,Loc,T),move(ferry,From,To,T),unboard(Car,Loc,T).

%state domain
state(0..steps).

%Action choice

{board(Car,Loc,T)}:-
    empty(ferry,T),
    at(Car,Loc,T),
    at(ferry,Loc,T),
    not affected(ferry,T),
    not goal(T).

{unboard(Car,Loc,T)}:-
    in(ferry,Car,T),
    at(ferry,Loc,T),
    not affected(ferry,T),
    not goal(T).

{move(ferry,From,To,T)}:-
    at(ferry,From,T),
```

```
    From != To,
    not goal(T).

%constraints

:- 2{at(Car,Loc,T):location(Loc),
     in(ferry,Car,T)},time(T),car(Car).

:- 2{at(ferry,Loc,T):location(Loc)},time(T).

:- 2{in(ferry,Car,T):car(Car)},time(T).

%identified affected objects

affected(Car,T):- board(Car,Loc,T).

affected(Car,T):- unboard(Car,Loc,T).

affected(ferry,T):-
     move(ferry,From,To,T),
     From != To.

%effects

in(ferry,Car,T+1):-
    board(Car,Loc,T).

at(Car,Loc,T+1):-
    unboard(Car,Loc,T).

at(ferry,To,T+1):-
    move(ferry,From,To,T).

%frame axioms

at(ferry,Loc,T+1):-
    at(ferry,Loc,T),
    not affected(ferry,T).

at(Car,Loc,T+1):-
    at(Car,Loc,T),
    not affected(Car,T).

in(ferry,Car,T+1):-
    in(ferry,Car,T),
    not affected(Car,T).

%other predicates
empty(ferry,T):-
     not in(ferry,Car,T).
```

E Encoding of the Elevator Problem in Smodels

Encoding of instance : initial and goal states

```
% objects
person(p).
floor(1).
floor(2).
above(2,1).

%initial state
at(p,1,0).
at(lift,1,0).
stop_at(lift,1,0).

%goal predicate
goal(T):-
    state(T),
    at(p,2,T).

goal(T+1):-goal(T).
goal:-goal(T).
compute {goal}.
```

Encoding of action system

```
%Written by Xiumei Jia

#domain person(P),state(T),floor(F),floor(From),floor(To).

#hide.
#show board(P,F,T),unboard(P,F,T),stop(lift,F,T).
#show up(lift,From,To,T),down(lift,From,To,T).

%state domain
state(0..steps).

%action choices

{board(P,F,T)}:-
   at(P,F,T),
   at(lift,F,T),
   not affected(lift,T),
   not goal(T).

{unboard(P,F,T)}:-
   in(lift,P,T),
   at(lift,F,T),
   not affected(lift,T),
   not goal(T).
```

```
{up(lift,From,To,T)}:-
   above(To,From),
   at(lift,From,T),
   not goal(T).

{down(lift,From,To,T)}:-
   above(From,To),
   at(lift,From,T),
   not goal(T).

%constraints

:- 2{at(lift,F,T):floor(F)},state(T).

:- 2{at(P,F,T):floor(F),in(lift,P,T)},
     state(T),person(P).

%Identify affected objects

affected(P,T):- unboard(P,F,T).

affected(P,T):- board(P,F,T).

affected(lift,T):- down(lift,From,To,T).

affected(lift,T):- up(lift,From,To,T).

%effects

at(lift,To,T+1):- down(lift,From,To,T).

at(lift,To,T+1):- up(lift,From,To,T).

in(lift,P,T+1):- board(P,F,T).

at(P,F,T+1):- unboard(P,F,T).

%frame axioms

at(P,F,T+1):- not affected(P,T), at(P,F,T).

in(lift,P,T+1):- not affected(P,T), in(lift,P,T).

at(lift,F,T+1):- not affected(lift,T), at(lift,F,T).
```

References

[1] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledhe for planning. *Artificial Intelligence*, 116(1,2):123–191, 2000.

[2] F. Bacchus, H. Kautz, D.E. Smith, D. Long, H. Geffner, and J. Koehler. AIPS-00 Planning Competition. In *http://www.cs.toronto.edu/aips2000/*, 2000.

[3] A. Barrett and D. Weld. Partial-order planning: evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71-112, 1994.

[4] P. V. Beek and X. Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the 16th National Conference on Artificial Intelligence, Orlando, Florida*, pages 585–590, July 1999.

[5] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI95)*, pages 1636–1642, 1995.

[6] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:161–204, 1994.

[7] W. Chen and D. S. Warren. Computation of stable models and its integration with logical query processing. *Knowledge and Data Engineering*, pages 742–757, 1995.

[8] P. Doherty and J. Kvarnstom. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. *TIME'99*, pages 47–54, 1999.

[9] K. Erol, D. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75–88, 1995.

[10] W. Faber and G. Pfeifer. http://www.dlvsystem.com/. Technical report, Vienna University of Technology, Computer Science Department, Institute of Information Systems (184), Database and Artificial Intelligence Group, 1996.

[11] R. E. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189-208, 1971.

[12] A. Finzi, F. Pirri, and R. Reiter. Open world planning in the situation calculus. In *Proceedings of AAAI-2000*, 2000.

[13] A. Frisch, I. Miguel, and T. Walsh. CGRASS: A system for transforming constraint satisfaction problems. In *Proc. Joint Workshop of the ERCIM/CologNet area on Constraint Solving and Constraint Logic Programming*, pages 15–30, 2002.

[14] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, 1988.

[15] M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of logic programming*, 17:301–321, 1993.

[16] M. Gelfond and V. Lifschitz. Action language. *ETAI*, 3(16):623–630, 1998.

[17] I. P. Gent and B. M. Smith. Symmetry Breaking During Search in Constraint Programming, Research Report 99.02. Technical report, School of Computer Studies, University of Leeds, 1999.

[18] N. Gupta and D. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56:223–254, 1992.

[19] G. Huang, X. Jia, C. Liau, and J. You. Two-Literal logic programs and satisfiability representation of stable models: A comparison. In *Proc. 15th Canadian Conference on AI, LNCS, Springer*, pages 119–131, 2001.

[20] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[21] J. Laird, A. Newell, and P. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–67, 1987.

[22] H. J. Levesque, R. Reiter, and etc. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, pages 59–83, 1997.

[23] W. Marek and M. Truszczynski. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.

[24] I. Miguel. Symmetry-breaking in planning: schematic constraints. In *Proceedings of the CP'01 Workshop on Symmetry in Constraints (SymCon '01)*, pages 17–24, 2001.

[25] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79, 1998.

[26] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming,Bonn, Germany*, pages 289–303, September 1996.

[27] I. Niemelä and P. Simons. Smodels - An implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning, volume 1265 of Lecture Notes in Artificial Intelligence*, pages 420–429, July 1997.

[28] I. Niemelä and P. Simons. Extending the Smodels system with cardinality and weight constraints. In *Jack Minker, editor, Logic-Based Artificial Intelligence, pages 491-521, Kluwer Academic Publishers*, pages 491–521, 2000.

[29] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *In Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 317–331, 1999.

[30] S. Padmanabhuni. *Logic programming with stable models for constraint satisfaction.* PhD thesis, University of Alberta, 2000.

[31] P. Simons. Efficient implementation of the stable model semantics for normal logic programs. Technical Report Research Report 35, Helsinki University of Technology,Helsinki, Finland, September 1995.

[32] P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Technical report, Research Report A47,Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, 1997.

[33] P. Simons. *Extending and Implementing the Stable Model Semantics.* PhD thesis, Helsinki University of Technology, Helsinki, Finland, 2000.

[34] B. Smith. Reducing symmetry in a combinatorial design problem, Research Report 2001.01. Technical report, University of Leeds, 2001.

[35] T. C. Son, C. Baral, and S. McIlraith. Planning with different forms of domain-dependent control knowledge - an answer set programming approach. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 226–239, September 2001.

[36] T. Syrjänen. http://www.tcs.hut.fi/Software/smodels/. Technical report, Laboratory for Theoretical Computer Science (TCS), Department of Computer Science and Engineering, Helsinki University of Technology.

[37] T. Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report Research Report D18, Digital Systems Laboratory, Helsinki University of Technology, October 1998.

[38] M. Veloso. Learning by analogical reasoning in general problem solving. Technical report, 1992.

[39] M. Veloso and J. Carbonell. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), 1995.

[40] D. Wilkins and M. desJardins. A call for knowledge-based planning. *AI Magazine*, 22(1), 2001.